Software Developer's JOURNAL

new ideas & solutions for professional programmers

starterkit

Vol.2 No.13 Issue 13/2013 (19) ISSN 1734-3933

Python Starter Kit

Python start kit – all you have to know 1st day

Most common errors of beginner with Python

Python for Windows how to set up development environment?

Showing the way from beginner to real pro

jax 2013 LONDON
the best of Java tech

featuring BIG DATA CON LONDON 2013

October 28th – 30th 2013 | Park Plaza Victoria, London

15% off with discount code JL13SD

• 60+ sessions • Keynotes • Expo Area •
• Hands-on workshops • Community Events •

www.jaxlondon.com

Early Bird discount ends 15th August - £649 all access!

follow us: twitter.com/JAXlondon JAXlondon JAX London

continuent
tungsten 2.0

The best MySQL clustering and replication solution gets better!

- ✓ Ideal For Large-Scale SaaS Applications
  Automatic failover, zero down-time maintenance, no slave lag, and backup-and-restore integration
- ✓ True Multi-Master, Multi-Site
  Deploy Tungsten clusters across multiple data centers or cloud-computing regions with real-time updates and failure handling for HA and DR
- ✓ Real MySQL Multi-Region Solution for Amazon AWS
  Relying on Amazon RDS may result in significant downtime. Continuent Tungsten offers true HA and DR throughout and beyond Amazon's availability zones and regions
- ✓ "Oracle GoldenGate Without The Price Tag"
  Replication from Oracle to Oracle Oracle to MySQL, and MySQL to Oracle

continuent
continuent.com/software

Dear Readers,

Our 'Python Starter Kit' is prepared for you to start your adventure with Python Programming. We did our best to collect the best authors, who will show you how to start.

For the very beginning we have tutorials, which will teach you step by step how to begin. In Python Guide for Beginners by Mohit Saxena you will be introduced to Python as a programming language, Sotaya Yakubuin his Starting with Python will show you the best basics to good start with Python. All you have to know to open yourself possibilities in Django you will find in the article of Alberto Paro : Beginning with Django.

When you catch some basics you will be ready for a bit more advance subjects that our authors prepared for you learn how to make Better Django Unit Testing Using Factories instead of Fixtures with Anton Sipos. Another thing that is good to know is a library in the article Python Fabric by Renato Candido. W. Matthew Wilson will introduce you to Python logging module, after that read some about Web Security in Python and Django by Steve Lott, interesting and pleasant tutorial about why safety is so important.

You will know more and more about Python, and thanks to George Psarksis and his Building a console 2-player chess board game in Python you will be able to learn Python Object-Oriented Concepts Interacting with user input on the command line. If you want to know how to write an app you should read Write a Web App and Learn Python by Adam Nelson.

For those who feel they want more, we have excellent article by Yves J. Hilpisch : Efficient Data and Financial Analytics with Python, which will make you be able to face today's data analytics challenges.

Our tutorial: Test-Driven Development With Python by Josh VanderLinden will guide you through mysteries of testing, with such great commented and done guideline you will really enjoy learning. For the end get to know what Python Interetors are and how Saad Bin Akhlaq is showing you its secrets.

I hope you will enjoy learning with us! For more take a look at our Python Programming issue, and enjoy what Software Developer's Journal prepared for you!

For staying in touch follow us on twitter and like us on facebook.

Karolina Rekun
& the SDJ Team

Software Developer's Journal
new ideas & solutions for professional programmers

team

Editor in Chief: Karolina Rekun
karolina.rekun@software.com.pl

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

Managing Director: Ewa Dudzic

Production Director: Andrzej Kuca
andrzej.kuca@sdjournal.org

Art. Director: Ireneusz Pogroszewski
ireneusz.pogroszewski@sdjournal.org

DTP: Ireneusz Pogroszewski

Marketing Director: Ewa Dudzic

Publisher: Hakin9 Media SK
02-676 Warsaw, Poland
Postepu 17D
Phone: 1 917 338 3631
http://en.sdjournal.org/

Python: A Guide for Beginners

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.

Figure 1. *Python Logo*

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 of 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 there is 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).

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 get familiarized 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).

Getting started with Python

As Python is an interpreted language, therefore 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

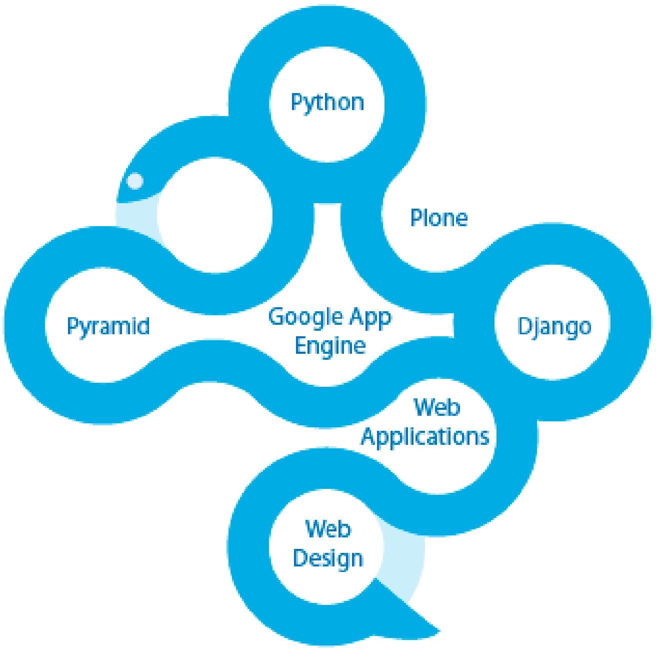


Figure 4. *Python and other similar languages*

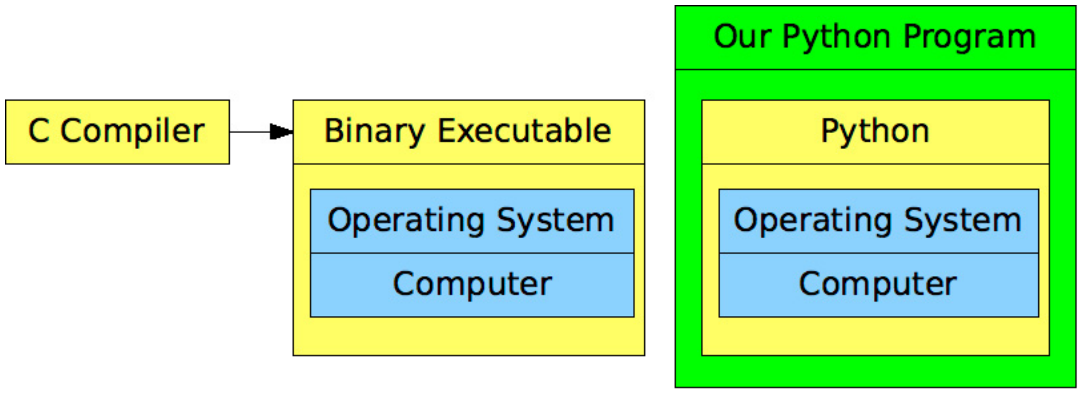


Figure 2. *How the computer sees Python*

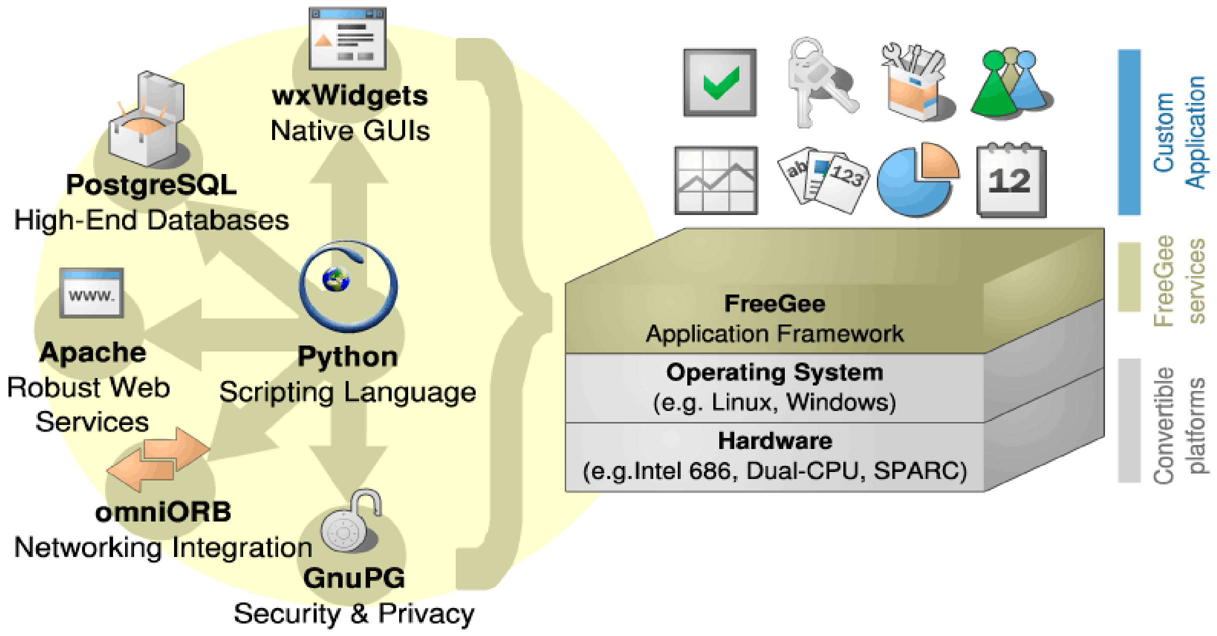


Figure 3. *System requirement for Python*

website content since the past five years. Each team member has at least two years of experience in writing for web.

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).

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:

Listing 1. *Simple code example of Python*

```
// def insert_powers(numbers, n)
//   powers = (n, n*n, n*n*n)
//   numbers [n] = powers
//   return powers

static PyObject  *
insert_powersl(PyObject *self, PyObject *args)
{
   PyObject *numbers:
   int  n:

   if (!PyArg_ParseTuple(args, 'oi" , &numbers, &n)) {
    return NULL;
}
    PyObject  *powers = Py_BuildValue("(iii)" , n,
                  n*n, n*n*n);

  //Equivalent to Python: numbers[n] = powers
 if (PySequence_SetItem(numbers, n, powers) < 0) {
  return NULL;
   }

  return powers;
}
```

```
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 what you call 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
```

References

[1] *https://www.google.com/url?sa=i&rct=j&q=&esrc=s&source=images&cd=&cad=rja&docid=Uu27A3md38FOsM&tbnid=ygia7G_YS151YM:&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/p1c5-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=667&sout=0&tbm=isch&tbnid=qwB5Xw9W8VEtWM:&imgrefurl=http://quintagroup.com/services/python/applications&docid=sKv9o-jtWEP8pM&imgurl=http://quintagroup.com/services/python/python-applications.png&w=377&h=205&ei=wYPuUbuWDc-ciQeTtYGgCw&zoom=1&ved=1t:3588,r:77,s:300,i:235&iact=rc&page=17&tbnh=164&tbnw=301&ndsp=22&tx=223&ty=97*

[5] *https://encrypted-tbn2.gstatic.com/images?q=tbn:ANd9GcRNM0MYpdUcHbhV5hlRKv8nkEnsAKwNNukK9-1FhyFfbsoh07ra4g*

In the above example, the machine first calculates 2 * 3 and then adds 1 to it. The reason is multiplication is on the high priority (3) and addition is at the below priority (4). In other 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/*.

MOHIT SAXENA

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

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, 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:

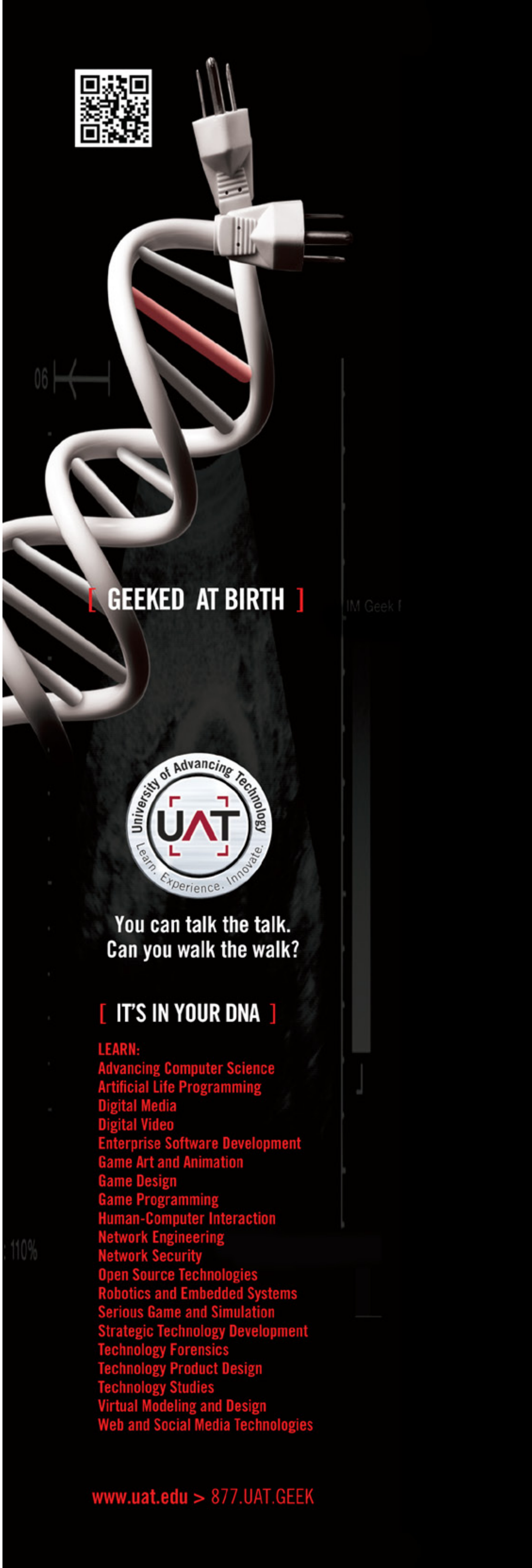
```
>>> div = 5/2
>>> print div
2
```

I know you want to ask a question, how did *5/2* becomes 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.

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.


[GEEKED AT BIRTH]
University of Advancing Technology
UAT
Learn. Experience. Innovate.
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

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.

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.*

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 throught each and eveluate

Listing 1. *Division and modular*

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

Listing 2. *Importing individual functions*

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

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).

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 does whenever its needed.

I will show you how a function a written:

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

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

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

It prints out what you passed 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 reading you code know what you were doing and sometimes its helpful when you come back to modify you code or update. Comments in python are

Listing 3. *Adding elements to a list*

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

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
```

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 """
```

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
```

striped out during parsing, and we comment in python by putting # before the line we want to be commented. 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.

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.

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()`.

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 being 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.

SOTAYA YAKUBU

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.

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__', '__getattribute__', '__hash__',
            '__init__', '__module__', '__name__', '__new__', '__reduce__', '__reduce_ex__',
            '__repr__', '__self__', '__setattr__', '__str__']
```

INSPIRATION STARTS IN THE SKIES.

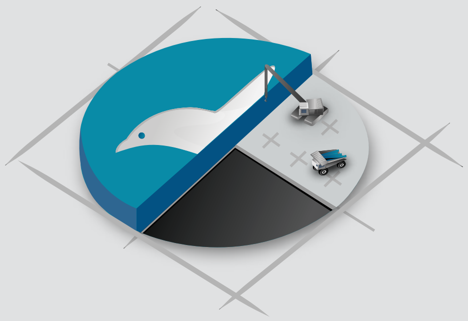
CIRIS
informatic solutions

Our **software development** service on the **cloud** will provide the security of having an updated, fast, and available software at all times without technical concerns. Just relax, enjoy your hazzle-free software!

Scala mongoDB

Outsourcing

Want to start using these technologies? Well, Ciris offers you years of experience in **Scala and MongoDB** to successfully guide you through this process.



Email us for a free consultation:
sales@ciriscr.com

www.ciriscr.com

Beginning with Django

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 dispachting), 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 that 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 to 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 others 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 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 to the virtual environment in 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

aThe 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 need to be themed or customized by users, so if generally overwritten with custom ones.

To create a new application is 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 to create missing tables and populating the initial database with the syncdb command.

```
python manage.py syncdb
```

The first time that is 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 will be created them with sequences, indices and foreign key contraints.

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.

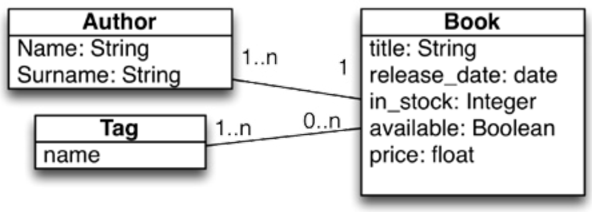


Figure 1. *Creating a book shop*

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

The models.py file

Create a Django Model is very easy: every model derives from models.Model and it's 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 shoud use NullBooleanField
- FloatField used to store floating point values
- ForeignKey used to store reference to others 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 extends them so 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_lenght (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 the 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).

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

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))
```

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')
)
```

Response object or its derived ones. We need to defines two views "index" and "detail" (Listing 3).

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 queyset, 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 a 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).

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 translate in 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.

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

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})
```

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

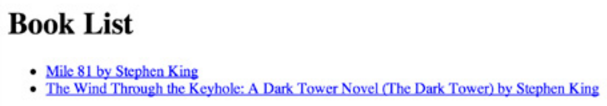
```
<!DOCTYPE html>{% load i18n %}
<html><head><title>{% trans "Index of books"
                  %}</title></head>
<body>
<h1>{% trans "Book List" %}</h1>
<ul>
{% for book in books %}
    <li><a href="{% url "shop:detail" book.pk
                  %}">
        {{ book.title }} {% trans "by" %} {{
                  book.author }}</a>
    </li>
    {% empty %}
    <li>{% trans "No books" %}</li>
    {% endfor %}
</ul>
</body>
</html>
```

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

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 every serious application is to have an admin interface in which insert/edit/delete your application data. Django, using reflection, allows create simple admin interface with few lines of code.



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*

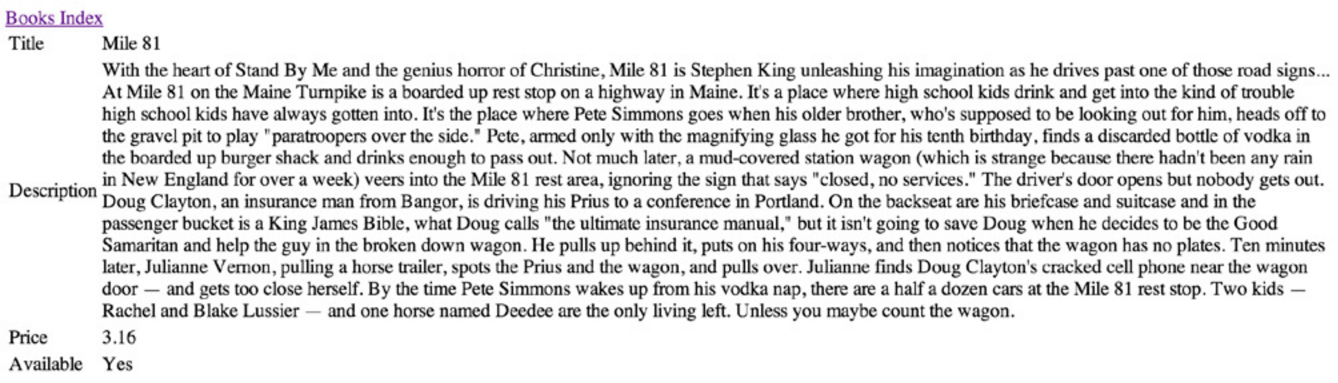
To activate the admin interfaces, the admin module discovery and the admin urls must be registered in the main urls file (mybookstore/urls.py) (Listing 6).

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

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



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*

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

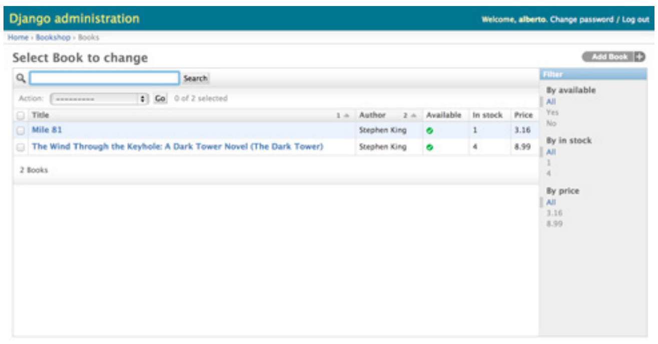
```
<!DOCTYPE html>{% load i18n %}
<html><head><title>{% trans “Book” %} - {{ book.
                 title }}</title></head>
<body>
<a href=”{% url “shop:index” %}”>{% trans “Books
                 Index” %}</a>
<table>
    <tr>
        <td>{% trans “Title” %}</td><td>{{ book.
                 title }}</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>
```

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)),
)
```

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)
```


Django administration

Welcome, alberto. Change password / Log out

Home › Bookshop › Books

Select Book to change

Title	Author	Available	In stock	Price
Mile 81	Stephen King	✓	1	3.16
The Wind Through the Keyhole: A Dark Tower Novel (The Dark Tower)	Stephen King	✓	4	8.99

2 Books

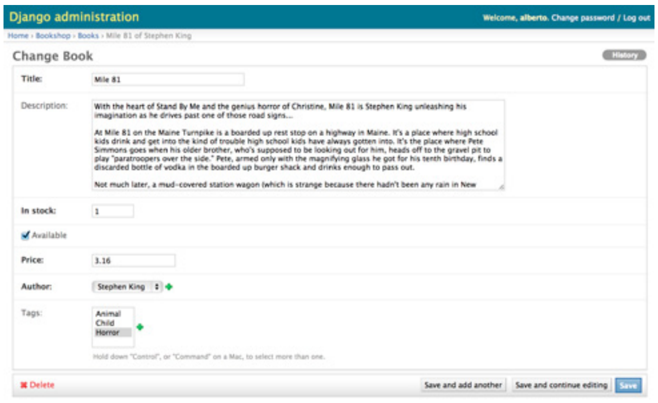
Filter

By available
All
Yes
No

By in stock
All
1
4

By price
All
3.16
8.99

Figure 4. *Django Admin – Book List Page*


Django administration

Welcome, alberto. Change password / Log out

Home › Bookshop › Books › Mile 81 of Stephen King

Change Book

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

In stock: 1

✓ Available

Price: 3.16

Author: Stephen King

Tags: Animal, Child, Horror

Hold down "Control", or "Command" on a Mac, to select more than one.

Figure 5. *Django Admin – Book Add-Edit Page*

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

- `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).

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 a lot of many others such as the cache, user/group management, middlewares, custom filter and tags, …

ALBERTO PARO

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

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 these tests easier using model factories instead of Django's data fixtures.

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

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.

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.
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 a 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 standard 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.

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.

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, andname 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 autogenerated data.

We don’t control this data (although as we’ll see later, we can tell model-mommy how to generate these

Easy and Quick Download of YouTube Videos!

WiseVideoSuite
Wise Video Downloader 1

- ✓ Access to HD Videos
- ✓ Search Without Browsers
- ✓ A-link-is-enough Download
- ✓ Easy and Fast

Official Website for More Information:
www.wisevideosuite.com/wisevideodownloader.html

Compatible with Windows 7 · Windows 8 Compatible

Support system:
Windows XP, Vista, Win7/8
(both 32-bit and 64-bit)

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, model-mommy will end up creating new Users with hometown fields automatically populated. You only need to specify a hometown in tests that make as-

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()
```

Listing 3. *Model of data*

```
[
    {
        "fields": {
            "description": "TestDesciption",
            "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
    },
]
```

sertions 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'
```

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()
```

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')
```

Listing 6. *Specifing 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'
```

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
'MhInizJgWlLYrNFVkxRgsTyOXHHaO
fqhrHrQbeGRADBEjzBTJI'
```

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 you want specified: Listing 6.

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

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)
```

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')
```

keys. Suppose we wanted be able to create multiple posts, all with unique dates and unique users. We could do this as follows: Listing 7.

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 is very easy to do with model mommy: Listing 8.

In this example we used model-mommy's shortcut of passing the '_quantity' argument to mommy.make to create many models as 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.

ANTON SIPOS

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.

▼ INSIGHTS FROM THE EXPERTS

JOHAN ARWIDMARK

SEAN DEUBY

CHANDER DHALL

MARY JO FOLEY

DAN HOLME

MARK MINASI

MICHAEL OTEY

MARK RUSSINOVICH

NEW CONTENT – GREAT NETWORKING – FUN!

IT/Dev CONNECTIONS

Las Vegas
Mandalay Bay
Sept 30 – Oct 4, 2013

For more than 13 years, IT/Dev Connections has been the premier training event for developers and IT professionals. IT/Dev Connections provides in-depth training on the technology platforms you're currently using, real-world solutions that will give you the competitive edge, and expert insight into how to plan for and implement the latest technologies. With more than 175 sessions to choose from, the conference offers training on HTML5, ASP.NET, Exchange, SQL Server, Windows OS, Windows Server, SharePoint, Visual Studio, Office 365, business intelligence, cloud, and all types of development.

Software Developer Journal Subscribers — use the code ITDSD13 and get

$200 off any of the registration packages!

BROUGHT TO YOU BY ▼

Penton · WindowsITPro · SQL Server PRO · DevPro · myITforum.com Powered by you · Paul Thurrott's SUPERSITE FOR WINDOWS · SharePoint Pro

Go to **devconnections.com** to register

Using Python Fabric to Automate GNU/Linux Server Configuration Tasks

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 to perform 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.

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.

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.

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`.

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 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

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')
```

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
```

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.
```

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.

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.

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.

Listing 4. *output of fab -l*

```
Available commands:

    local_info
    remote_info
```

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')
```

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:
[192.168.250.150] out: sudo password:

[192.168.250.150] out:

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

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')
```

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.
```

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.

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.

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.

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

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/*

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')
```

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')
```

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')
```

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()
```

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()
```

pep-0343/), to specify the `env` settings. A compatible `create_dir()` task using the with statement is shown on Listing 11.

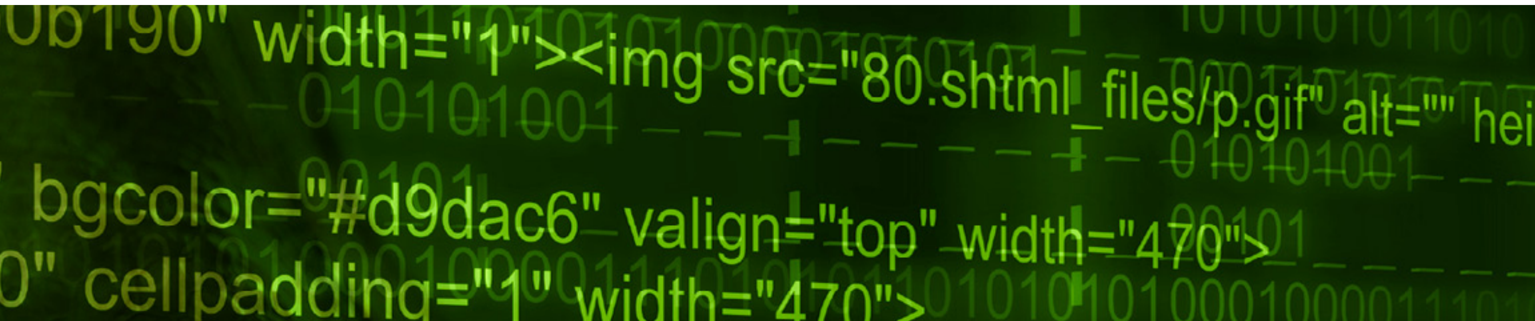
The `fab` command is useful for performing system administration and application deployment tasks from a shell console. However, sometimes you may want to execute tasks from within your Python scripts. To do this, you may simply call the Fabric functions from your Python code. To build a script that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

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.

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 learning a bit about Fabric to automate some of your tasks!

RENATO CANDIDO

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.



UPDATE
NOW WITH
STIG
AUDITING

"IN SOME CASES
nipper studio
HAS VIRTUALLY
REMOVED
the NEED FOR a
MANUAL AUDIT"

CISCO SYSTEMS INC.

Titania's award winning Nipper Studio configuration auditing tool is helping security consultants and end-user organizations worldwide improve their network security. Its reports are more detailed than those typically produced by scanners, enabling you to maintain a higher level of vulnerability analysis in the intervals between penetration tests.

Now used in over 45 countries, Nipper Studio provides a thorough, fast & cost effective way to securely audit over 100 different types of network device. The NSA, FBI, DoD & U.S. Treasury already use it, so why not try it for free at www.titania.com

2012 Computing Security Awards
WINNER
Enterprise Security Solution of the Year

Security Products' GOVIES
2013 Government Security Awards

2012 Computing Security Awards
WINNER
Network Security Solution of the Year

www.titania.com

The Python Logging Module is Much Better Than Print Statements

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

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*

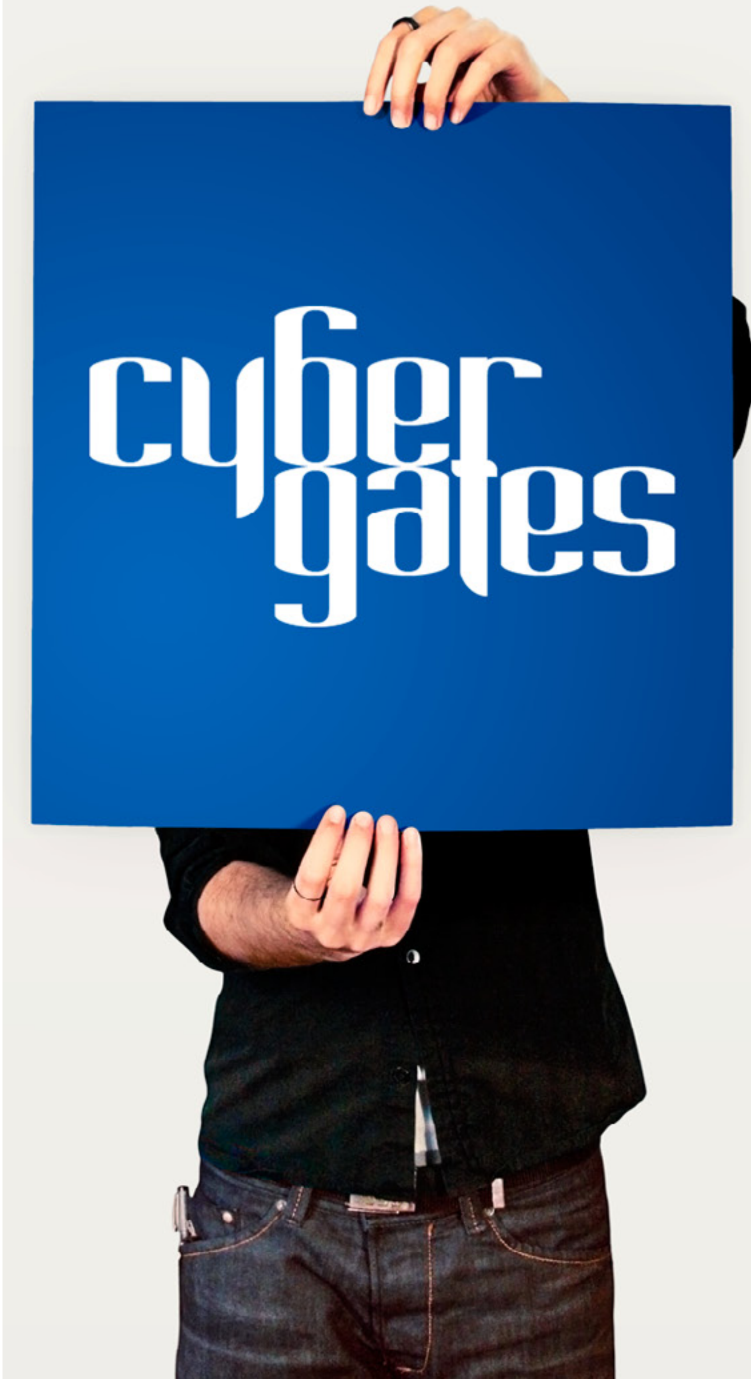
```
# This is b.py.
import logging

# 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()
```



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 gets or posts, you can send UDP packets, you can write to a local file, etc.

W. MATTHEW WILSON

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.

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()
```

A BZ Media Event

Big Data gets real at Big Data TechCon!

Discover how to master Big Data from real-world practitioners – instructors who work in the trenches and can teach you from real-world experience!

Come to Big Data TechCon to learn the best ways to:

- Collect, sort and store massive quantities of structured and unstructured data
- Process real-time data pouring into your organization
- Master Big Data tools and technologies like Hadoop, Map/Reduce, NoSQL databases, and more

Over 60 how-to practical classes and tutorials to choose from!

- Learn HOW TO integrate data-collection technologies with analysis and business-analysis tools to produce the kind of workable information and reports your organization needs
- Understand HOW TO leverage Big Data to help your organization today

"Big Data TechCon is loaded with great networking opportunities and has a good mix of classes with technical depth, as well as overviews. It's a good, technically-focused conference for developers."
—Kim Palko, Principal Product Manager, Red Hat

"Big Data TechCon is great for beginners as well as advanced Big Data practitioners. It's a great conference!"
—Ryan Wood, Software Systems Analyst, Government of Canada

"If you're in or about to get into Big Data, this is the conference to go to."
—Jimmy Chung, Manager, Reports Development, Avectra

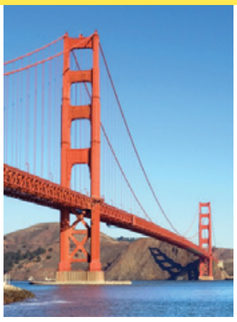
BigData TECHCON

San Francisco

October 15-17, 2013

www.BigDataTechCon.com

The HOW-TO conference for Big Data and IT professionals



Python, Web Security and Django

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. Details very, 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 hours 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).

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 classs 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.

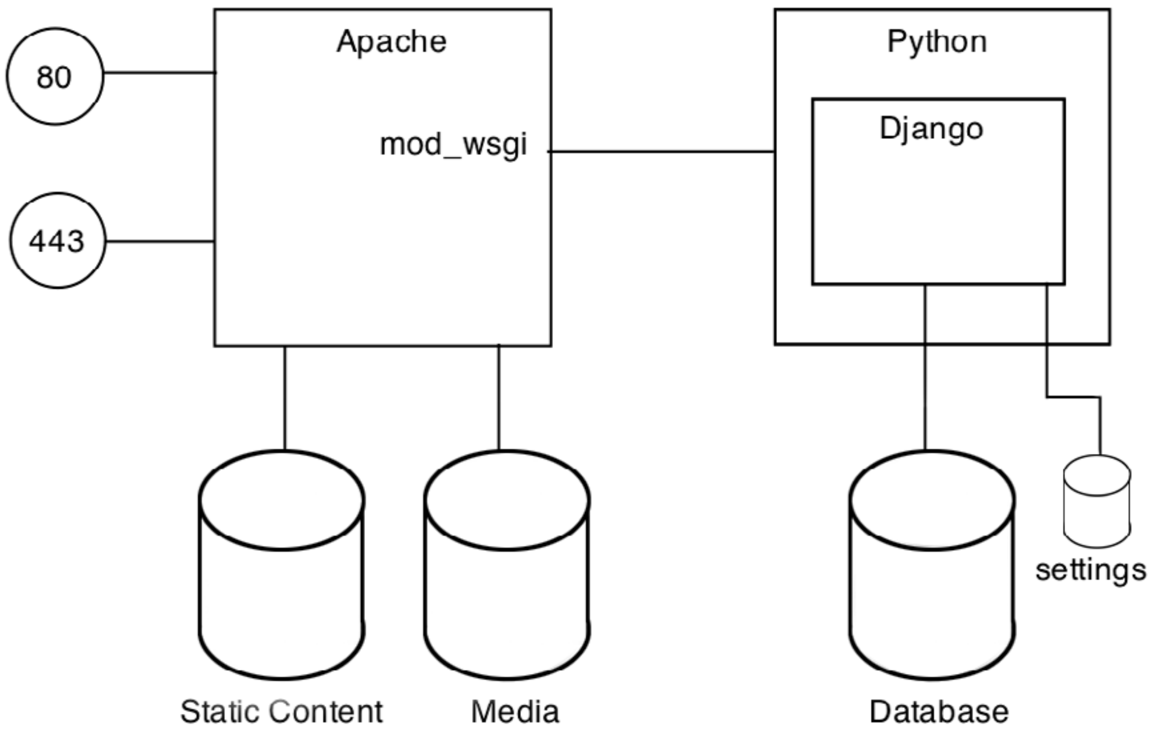


Figure 1. *The Apache and Django Architecture*

Additional User Features

Django's pre-1.5 `auth.profile` module can be used to provide all of 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 have 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 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 that 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 this 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.

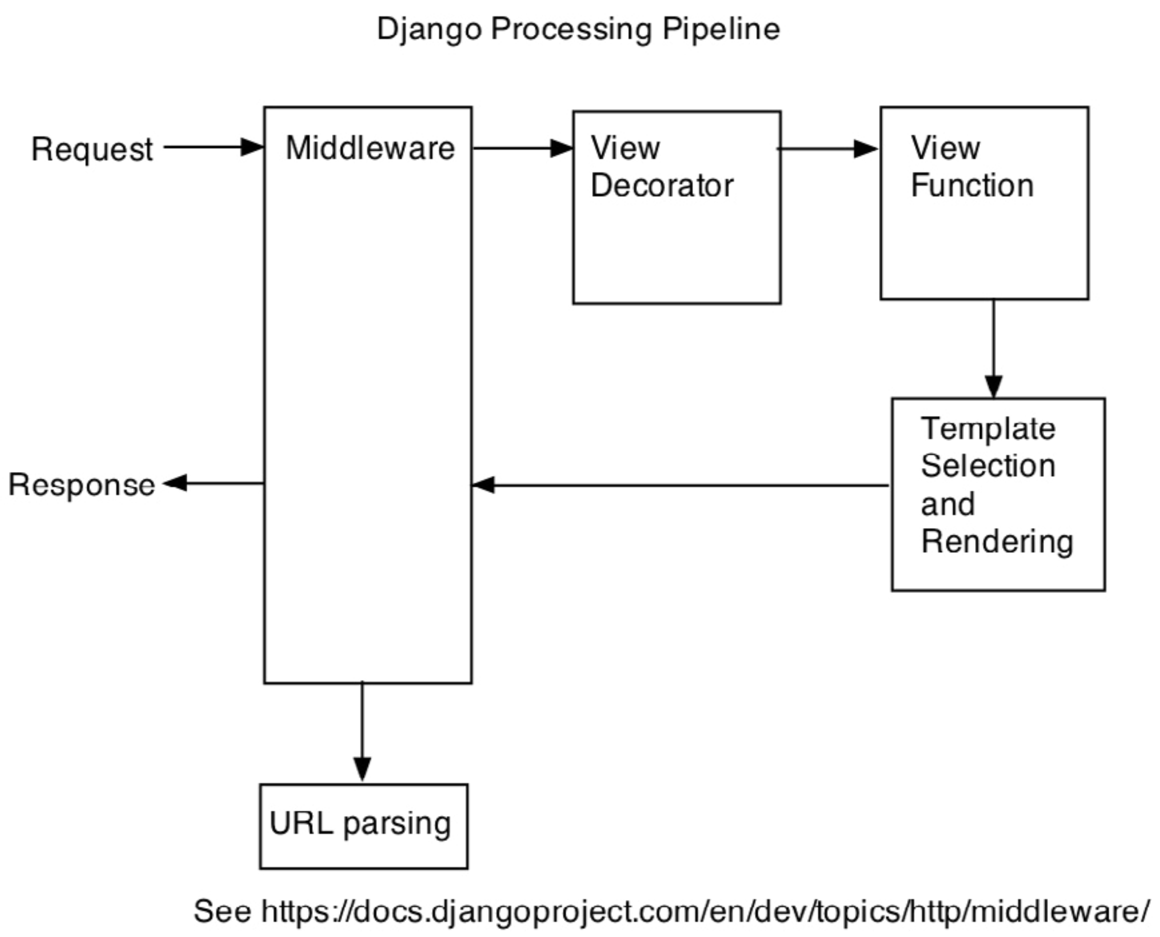

Django Processing Pipeline
Request
Middleware
View Decorator
View Function
Template Selection and Rendering
Response
URL parsing
See https://docs.djangoproject.com/en/dev/topics/http/middleware/

Figure 2. *Django Processing Pipeline*

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).

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` shold 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.

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

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)
```

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 several 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, 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.

STEVE LOTT

The author 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.

KPMG

cutting through complexity

Are you prepared?

kpmg.ca/forensic

INTRUSION

ATTACK • THREAT • CYBER SECURIT

TECHNOLOGY • CORPORAT

ELECTRONIC • INFORMATION • COMPLEXI

DATA ANALYTICS

RISK • INFORMATION • TECHNOLOG

DATA RECOVERY

COMPLEXITY • ELECTRONIC • INFORMATI

FORENSICS

DATABASE • ELECTRONIC • CONTRO

INTELLIGENCE

INFORMATION • RISK • TECHNOLOG

eDISCOVERY

COMPLEXITY • THREAT • INTELLIGEN

INVESTIGATIONS

TECHNOLOGY

COMPLEXITY • THREAT • DATABAS

INTELLIGENCE • PROTECTIO

CORPORATE

Building a Console 2-player Chess Board Game in Python

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 walk 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 hit "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 and
- 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.
- 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 simplistic 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 are making 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

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()
```

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).

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/PythonDecorator*s.

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:

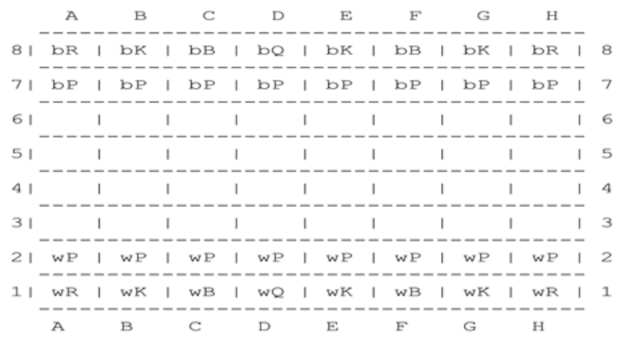
- A row at the top and the bottom containing the column names (A-H).
- Each row starts and ends with the row number.
- Squares are enclosed with "|" and "-" characters.
- Pieces are represented with their color's initial letter in lowercase and the initial letter of the piece type in uppercase. For example 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 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



```
     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*

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 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 if it 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 a 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 about 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* if 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*).

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 walked 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.

GEORGE PSARAKIS

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.

Dr.WEB®
20 years

Dr.Web SpIDer is 8-legged!

New Package Installer
Improved Technology for Protection against yet-unknown Threats
New Anti-Rootkit Module
Online Service Dr.Web Cloud
Improved Parental Control
Free Upgrade to New Version
Unified User Interface
Better than ever: Dr.Web Security Space version 8.0! 30-day trial available from https://download.drweb.com/?lng=en

New Version 8.0

Security Space and Dr.Web Antivirus for Windows

Get your free 60-day license under https://www.drweb.com/press/
to protect your PC and your smartphone with Dr.Web!
*Your promo code: **Hakin9***

Protect your mobile device free of charge!
https://support.drweb.com/free_mobile/

Doctor Web is a Russian anti-virus vendor with a software development record dating back to 1992.
www.drweb.com

Write a Web App and Learn Python

Background and Primer for Tackling the Django Tutorial

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.

a d v e r i s e m e n t

ModelOff 2013
The Ultimate Financial Modeling Challenge
$30,000 First Prize
Register NOW at www.modeloff.com
S&P CAPITAL IQ
Microsoft
Bloomberg INSTITUTE
MODELOFF FINANCIAL MODELING WORLD CHAMPIONSHIPS 2013

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 en-compassing, 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.

ADAM NELSON

ANRC

A Cyber criminal can target and breach your organization's perimeter in less than a second from anywhere in the world ...

Are You Prepared?

ANRC delivers advanced cyber security training, consulting, and development services that provide our customers with peace of mind in an often confusing cyber security environment. ANRC's advanced security training program utilizes an intensive hands-on laboratory method of training taught by subject matter experts to provide Information Security professionals with the knowledge and skills necessary to defend against today's cyber-attacks and tomorrow's emerging threats.

ANRC's consulting and development services leverage team member knowledge and experience gained in the trenches while securing critical networks in the U.S. Department of Defense and large U.S. corporations. ANRC tailors these services to deliver computer security solutions specific to the needs of the customer's operational environment. Our approach emphasizes a close relationship with our clients as an integral part of our service. We believe we're all in the security battle together, and we view our customers as key members of our team in the fight.

TRAINING :: CONSULTING :: SOLUTIONS **www.anrc-services.com**

Efficient Data and Financial Analytics with Python

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 why it makes it so much popular in the world of programmers.

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 where 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.

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*)

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 a 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 a 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 considered:

- departments: the majority of companies is organized in 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 shall 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 is 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

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

where z is a standard normally distributed random variable and it holds $0 < t \le T$ 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 especially 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).

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

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)
```

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
```

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()
```

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).

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).

The result from this code is shown in Figure 1 with the thicker red line being the average over all paths.

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 especially 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
```

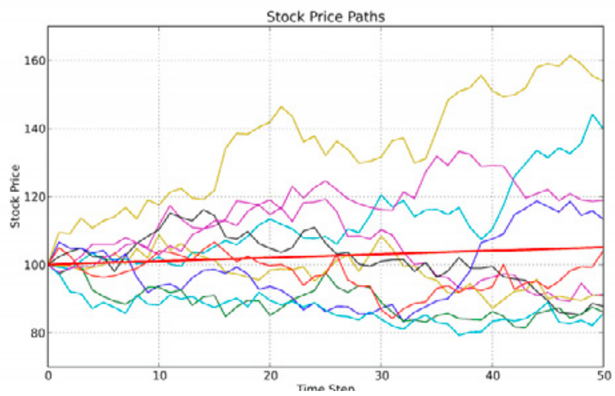


Figure 1. *10 simulated stock price paths and the average over all paths (red line)*

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 two other 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.

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:
```

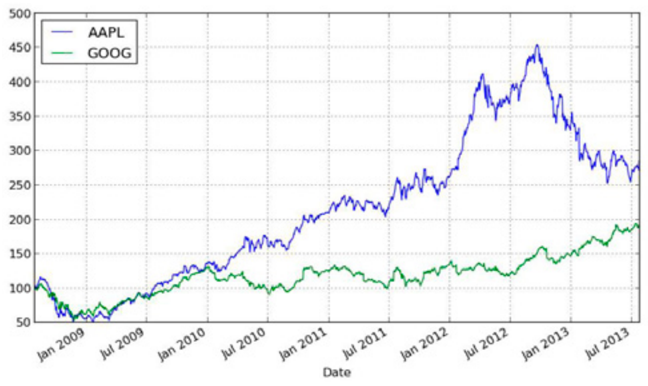


Figure 2. *Apple and Google stock prices since 28. July 2008 until 02. August 2013; both time series normalized to start at 100*

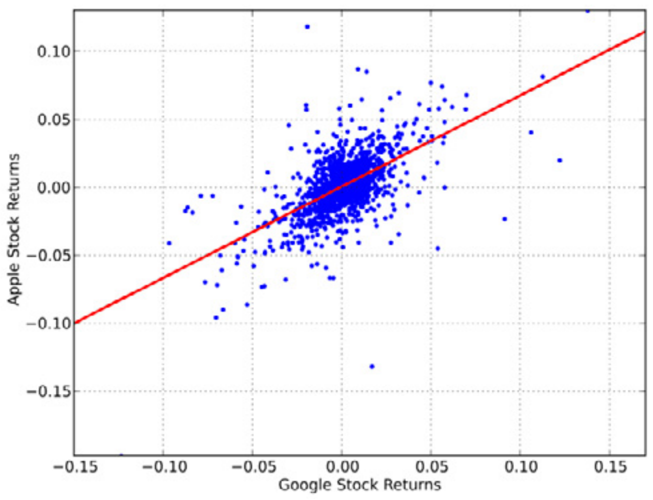


Figure 3. *Scatter plot of Google and Apple stock price returns from 28. July 2008 until 02. August 2013; red line is the OLS regression result with y = 0.005 + 0.67 x*

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).

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/ma-

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')
```

chines. 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.

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 by 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 x 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 in 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 in 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 size 700 MB 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.

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 otherwise on the given hardware beforehand.

Finally, you should close your database.

```
In:
h5.close()
```

The example illustrates that PyTables allows to implement 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 what 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.ear.read()
h5.close()
```

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)
```

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.

DR. YVES J. HILPISCH

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

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 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.

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 1. *An empty TestCase subclass*

```
from unittest import TestCase


class GameTest(TestCase):

    pass
```

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'))
```

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 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.

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.

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

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)
```

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
```

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)
```

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.

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).

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).

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).

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

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)
```

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. *Implemention meets our expectations*

```
..
------------------------------------------------------
Ran 2 tests in 0.003s

OK
```

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
            }
        )
```

forget to stub out the new `Pig.get_scores` method (Listing 13 and Listing 14).

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).

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).

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 known input.

Testing the internal implementation of a function or method is asking for 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 unit 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 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.

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)
```

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
```

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
```

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 want to 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 that 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).

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 as been executed, the context manager will roll back the `input` function to its

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
```

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)
```

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.

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).

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).

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 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
```

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: ")
        ])
```

Listing 26. *All tests pass*

```
.....
------------------------------------------------------
Ran 5 tests in 0.005s

OK
```

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.

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.

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')
```

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, **keywargs)
  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
```

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).

We've added a `setUp` method to our class. This method name has a special meaning when used with Py-

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()
```

thon'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).

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.

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')
```

Listing 36. *Refactoring has not broken our tests*

```
......
----------------------------------------------------------------------
Ran 6 tests in 0.005s

OK
```

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
            }
        )
```

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.

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.

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

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, **keywargs)
  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
```

on the fact that our assertions up to this point are passing. bSo let's take a look at our failing test (again, after stubbing the new `play` method): Listing 38.

Marvelous, the test fails, exactly as we want 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.

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 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, **keywargs)
  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
```

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 a 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).

Let's see if the test output has been cleaned up at all with our updated test (Listing 45).

Isn't `mock` wonderful? It is so 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 con-

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
```

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
            }
        )
```

Listing 45. *All tests pass with no print output*

```
.......
------------------------------------------------------

Ran 7 tests in 0.007s

OK
```

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!')
```

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, **keywargs)
  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 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
```

tinue 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).

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. nWe'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)?

Hey, there's the `StopIteration` exception that we discussed a couple of times before. We've only specified two roll values, which should be just 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).

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. Why 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, **keywargs)
  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
```

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.

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` thus far have been 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.

Listing 51. *Players may now win the game*

```
........
------------------------------------------------------
Ran 8 tests in 0.009s

OK
```

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')
        ])
```

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, **keywargs)
  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)
```

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).

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!

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).

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
```

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.

JOSH VANDERLINDEN

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

Python makes a distinction between iterables and iterators, it is quite essential to know the difference between them. Iterators are stateful objects they know how far through their sequence they are. Once they reach their thats 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 lines (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 to a *for* loop and an iterator dictates what each iteration of the loop returns (Listing 1 and Listing 2).

Some types like 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).

Why the distinction? An iterable object is just something that it 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 which 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 t 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
```

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()
```

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__()
```

`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 fallvalue 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 is 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(lamdba 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.

SAAD BIN AKHLAQ

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.

a d v e r t i s e m e n t

Workbooks.com

Web Based CRM & Business Applications for small and medium sized businesses

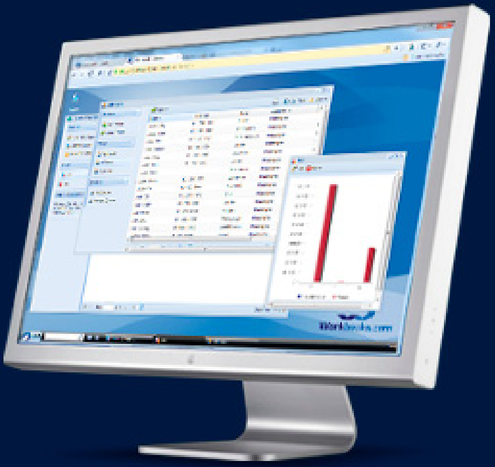
Find out how Workbooks CRM can help you

- Increase Sales
- Generate more Leads
- Increase Conversion Rates
- Maximise your Marketing ROI
- Improve Customer Retention

Contact Us to Find Out More

+44(0) 118 3030 100

info@workbooks.com



We're hiring!

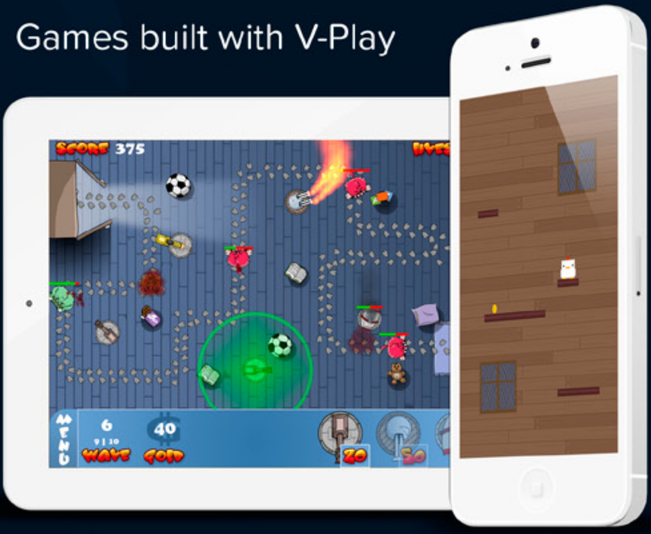
GAME ENGINE V-PLAY

making cross-platform game development easier

- ✓ In-game level editor
- ✓ Balance your game while it is running
- ✓ Give your players the power to create their own levels and keep them playing for longer
- ✓ Let players share their levels and build a community around your game
- ✓ Monetize user-generated content
- ✓ Specialized for 2D games
- ✓ Free V-Play SDK includes open-source game templates of popular genres
- ✓ Rapid component-based development with QML & JavaScript
- ✓ Plugins for in-app purchases, game networks, analytics and much more
- ✓ Professional toolchain includes Qt Creator IDE and V-Play Build Server

Games built with V-Play

SQUABY
Tower Defense

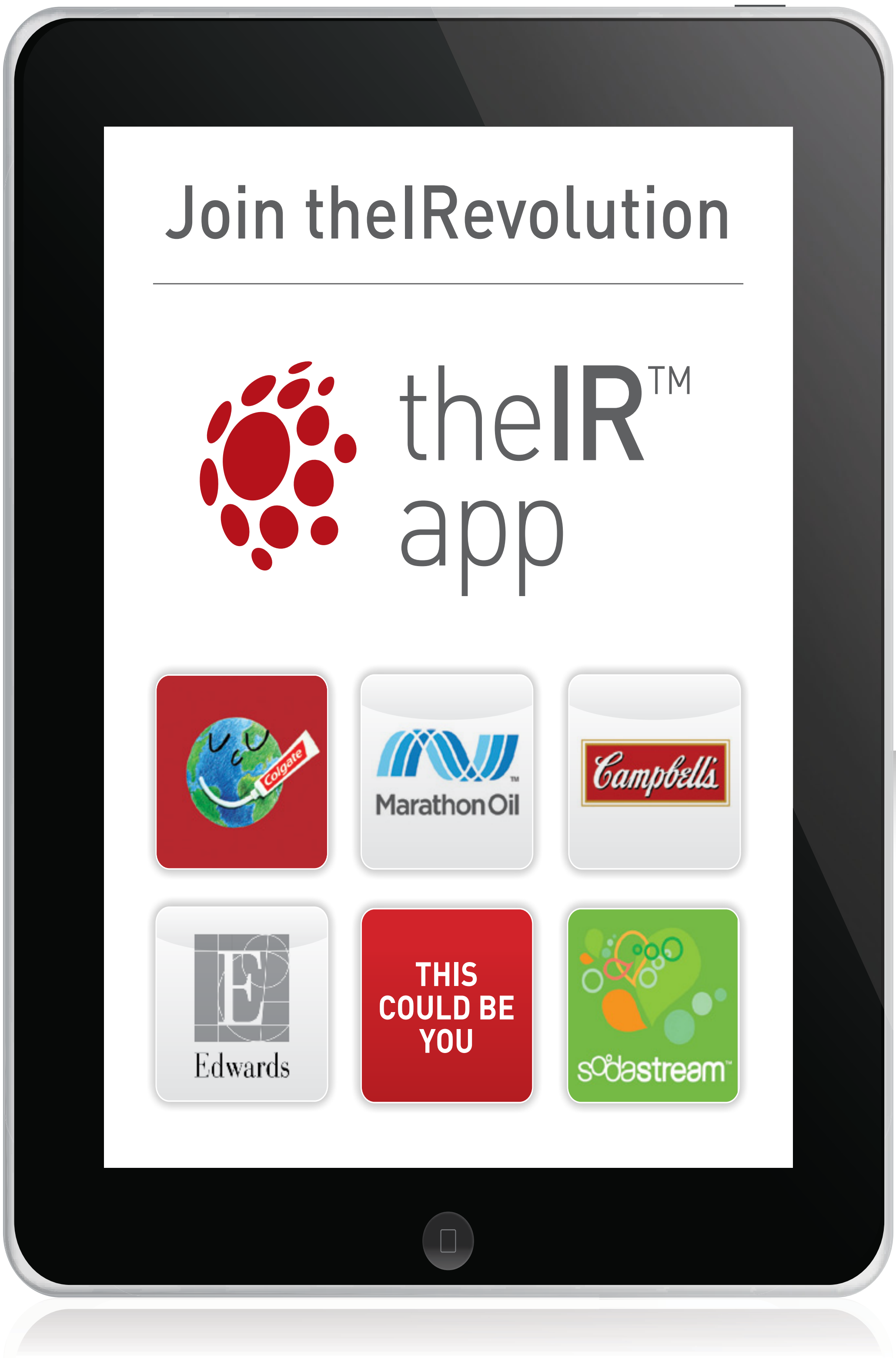


CHICKEN OUTBREAK
Platformer

for more showcase games see: http://v-play.net/showcases

iOS android BlackBerry symbian MeeGo DESKTOP

DON'T BE LEFT OUT



www.theIRapp.com