perjury Documentation

Release 0.0.1

Aaron Merriam

October 02, 2012

CONTENTS

Perjury is a suite of tools for content generations. The overall goal of this module is to both provide a thorough set of tools for generating fake content, and to make writing new generators or extending existing generators a simple process.

We want developers to spend time developing, rather than populating fake blod entries with fake content.

Contents:

CHAPTER

ONE

GETTING STARTED

Getting up and running with Perjury generators is very simple.

```
>>> from perjury.generators import smallint
>>> smallint()
3
>>> smallint()
500
>>> [smallint() for x in range(10)]
[48, 2, 71, 64, 3, 46, 99, 33, 80, 3]
```

Perjury is built around the concept of *callables*. All generators provided by Perjury implement the same simple interface.

Note: Calling the generator returns a value

Admittedly `perjury.generators.smallint()` is not a very impressive generator, so lets jump right into some real use cases. For this example we will be generating a generic user. In order to try and keep this generic, lets use the following schema for what our users will look like.

Note: Users of Django will likely recognize the following as a slightly mutated version of Django's build in User model. While trying to keep this example simple, we also want to showcase an example that solves a real world problem.

- User
  - username (unique)
  - hashed password
  - is_active
  - is_superuser

The fist thing we'll need is a username generator. While there are many ways to do this, we're going to to use a random choice from a predefined set of usernames.

```
import random

usernames = ['animal', 'beaker', 'fozzie', ... , 'scooter']
def username():
    return random.choice(usernames)
```

This is a pretty solid solution for most situations. However, in practice, we could quickly run into some issues with uniqueness across the set of generated usernames. Instead of diving down the rabbit hole of ways to implement uniqueness across our username function, lets use Perjury's built in function decorator `perjury.util.unique()`. The simplest implementation works for most cases.

```
>>> from perjury.util import unique
>>> [username() for a in range(3)]
['animal', 'fozzie', 'animal'] # can (and will) return repeat values
>>> unique_username = unique(username)
>>> [unique_username() for a in range(10)]
['animal', 'fozzie', ...] # will enforce uniqueness across return values.
```

Note: The unique decorator will *just work* in most cases. It is however important to read the full documentation on how it works, and how you can configure how uniqueness constraints are enforced.

Of course, Perjury comes with a username generator that should work for most needs. The point of this however is to illustrate that Perjury both provides a very broad set of tools to generate content, and can be used to very easily build your own generator for any kind of data.

We're going to skip over password for now and move onto our other fields. For our `is_active` boolean field, we will want to generate some random `True` and `False` values, but unevenly distributed. Lets generate 1 inactive user for every 3 active users.

```
import random

def active():
    return bool(random.randint(0, 3))
```

Perjury comes with an easy to use `weighted_choice()` generator. For now, we'll just use our `active` function.

Next, lets create our function for generating superuser status. Lets be sure that we only generate one superuser. While there are plenty of functional ways to do this, we'd like to take this time to demonstrate how you can easily write stateful generator callables.

```
from perjury.generators import BaseGenerator

class SuperuserStatus(BaseGenerator):
    superuser_generated = False

    def generator(self):
        if not self.superuser_generated:
            self.superuser_generated = True
            return True
        else:
            return False
```

And to use it.

```
>>> superuser_callable = SuperuserStatus()
>>> superuser_callable()
True
>>> superuser_callable()
False
>>> [superuser_callable() for i in range(5)]
[False, False, False, False, False]
```

So lets pull this all together into a cohesive user generator. Attentive readers will realize that we've left out a generator for the password field. For now however, lets look at what our user generator would look like combining all of our code so far.

```
import random
from perjury.util import unique

usernames = ['animal', 'beaker', 'fozzie', ... , 'scooter']

@unique
def username():
    return random.choice(usernames)

def active():
    return bool(random.randint(0, 3))

class SuperuserStatus(BaseGenerator):
    superuser_generated = False

    def generator(self):
        if not self.superuser_generated:
            self.superuser_generated = True
            return True
        else:
            return False

superuser_callable = SuperuserStatus()

def user_generator():
    user = User(username=username(), is_active=active(), is_superuser=superuser_callable())
    user.set_password('password')
    return user
```

Now you may see why we skipped over password. In our slightly fictional example model, instead of computing a password hash ourselves, it is much easier to use the built in API call to `set_password` to set the hashed password.

None of this is very novel at face value. Most programmers with a bit of experience could hammer out the code above in a short period of time. However, this code tends to be tedious at best and often involves a lot of 're-inventing the wheel' type of code. This is where Perjury comes in to save the day. Lets take a look at an implementation both functionally and class-based.

Functionally:

```
import itertools

from perjury.generators import username, weighted_choice
from perjury.util import unique

unique_username = unique(username)

def user_generator():
    user = User(
        username=unique_username(),
        is_active=weighted_choice({True: 1, False: 3}),
        is_superuser=itertools.chain([True], itertools.repeat(False))
        )
    user.set_password('password')
```

Class-Based:

```
import itertools

from perjury.generators import BaseGenerator, username, weighted_choice
```

```
class UserGenerator(BaseGenerator):
    unique = True
    key_fn = lambda u: u.username

    def generator(self):
        user = User(
            username=username(),
            is_active=weighted_choice({True: 1, False: 3}),
            is_superuser=itertools.chain([True], itertools.repeat(False))
            )
```

Perjury does its best to both provide a very broad set of tools, and ensure that its tools can be re-used and modified to suit your content generation needs. Most of the generators found in Perjury are build off of a small set of functional tools included with Perjury along with some thin wrappers around many of the tools python provides.

CHAPTER

TWO

BASE MODULE

`perjury.base` declares base generator classes used to constuct more useful generators.

2.1 Generators

CHAPTER

THREE

GENERATORS MODULE

`perjury.generators` provides some helpful common generator classes.

3.1 Generators

3.1.1 Name Generators

3.1.2 Text Generators

3.1.3 Content Generators

CHAPTER

FOUR

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

p