grampg Documentation

Release 0.2.0

Elvio Toccalino

August 06, 2015

Contents

Password generation with a fluent interface... a nice treat when you're under supervision of a grumpy sysadmin.

Contents:

CHAPTER 1

Description

The `grampg` code provides you with a **password generator object**, a bottomless barrel from which you can pull passwords. The difference between this *generator object* and something like `/dev/random` or `''.join([random.choice(letters) for i in range(LEN)])` is that you get to build that generator with great specificity, to obtain passwords adequate to your needs.

You will first have to build the generator, though. This could be a tedious, and a difficult task. Consider the following example:

```
how would you tell the builder you want passwords 10 characters long,
with lower letters and at least 3 numbers, starting with a letter?
```

`grampg` provides an easy to use interface, aimed at translating your needs intuitively.

Building the generator object is done in steps, using a **builder** object. At each step you add a *spec* (think of it as a piece of specification). You proceed working on the builder object accumulating *specs*, all of which add up to the requirements your passwords must satisfy. In the case of the example, you could use something like this:

```
gen = PasswordGenerator().of().length(10).at_most(10, 'lower_letters')
                                         .at_least(3, 'numbers')
                                         .beginning_with('lower_letters')
                                         .done()
```

The expression above will yield a generator object ready to produce passwords, exactly as you require them.

Note: The order in which *specs* are added is irrelevant. You are encourage to declare your *specs* as they sound more natural in your head; doing so greatly improves maintainability of the code.

Now that you have your generator object, you can use it throughout your own code:

```
passwords = [gen.generate() for i in xrange(so_many_passwords)]
```

And that's what `grampg` offers: a simple way to specify contrived password schema.

Note: In case you're wondering, passwords generated by the resulting generator object are **strong**. You can revise the algorithm later in this docs. XXX add reference to that.

1.1 The PasswordGenerator / Generator duality

Apart from a small hierarchy of three exceptions to deal with errors, the `grampg` exposes two classes to the user: the **Generator** and the **PasswordGenerator**. The naming might be confusing, but (as much in the design of this library) it is for the sake of code readability.

Of the two classes, the `grampg.Generator` is the actual password generator. It accumulates and holds your *specs*, generates the passwords and is the object which the user will keep reference to (in most, but not all, use cases). It implements the `grampg.Generator.generate()` method, and that says it all.

The `grampg.PasswordGenerator`, on the other hand, is the *builder* class. Its instances will create and stow away a `grampg.Generator` instance for the user, and will act as its interface as *specs* are added. It relinquishes control of the *generator object* only when the building phase is terminated (when it receives a call to `done()`).

This explains the naming choices: the user should never have the need to write `Generator()` but `PasswordGenerator.of()` (both idioms being equivalent, the second instantiates the generator object internally).

1.2 Technical description

The two classes exposed by the `grampg` module constitute a *builder* and *product* pair. The *builder* aids the user in specifying an adequate representation of the *product*.

Generators are instances of `grampg.Generator`. A generator object is instantiated with the sets of characters to use, and is responsible for accumulating the *specs* through method calls. The user, however, does not need to know any of that. The user never really interacts with a `grampg.Generator` instance directly, but through the builder. For the user, generator objects have one method of interest, `generate()`, which produces a single, independent and strong password string each time it's called, but which can be called only when the specification phase is *done*.

A builder object is an instances of `grampg.PasswordGenerator`, and provides a *fluent interface* to the Generator internal object being specified. The interface leverages *method chaining* and the *builder pattern* to provide a quick and easy specification phase. The `grampg.PasswordGenerator` provides a means of defining non-default character sets (characters to choose from when generating passwords), and passes user *specs* to the generator object being built. The builder object returns the generator object only when a call to `done()` succeeds. The generator object returned is ready to receive calls to `generate()`.

CHAPTER 2

`grampg` API

When interfacing the *grampg* you will instantiate *PasswordGenerator*, configure your character sets if necessary, and call *of()* on it to instantiate an internal *Generator*. The *PasswordGenerator* is in fact little more than a fluent interface to build generators. The generator instance is returned only when *done()* is called.

By means of the *PasswordGenerator* instance, the *Generator* instance can then be progressively spec'ed, so passwords generated by it can conform to you're twisted needs.

class `grampg.`**`PasswordGenerator`**(*from_sets={}*)

Build the password generator.

Provides a fluent interface to build *Generator* instances, by means of method chaining.

Exposes the character sets. Default character sets are provided for upper and lower case letters (*upper_letters* and *lower_letters*, respectively, all mashed up in *letters*) and *numbers*. A conjunction of the three is also provided, under the name *alphanumeric*.

A character set can be registered by keying its name to a list of eligible characters in the `sets` attribute, or by extending the default character sets during instantiation.

`at_least`(*low*, *setname*)

Spec method: require no less than *low* but no more than *high* characters from that set. This spec defines a range of characters.

`at_most`(*high*, *setname*)

Spec method: require no more than *high* characters from that set. This spec defines a range of characters.

`beginning_with`(*setname*)

Spec method: passwords will start with a char from this set.

Some other *spec method* must be called to define a number or range for that same set. Beginning with characters not specified is an error.

`between`(*low*, *high*, *setname*)

Spec method: require no less than *low* but no more than *high* characters from that set. This spec defines a range of characters.

`done`()

Finalize the generator and return it.

The returned instance can receive calls to *generate()*, each of which will produce an independent password complying with the specs.

`ending_with`(*setname*)

Spec method: passwords will end with a char from this set.

Some other *spec method* must be called to define a number or range for that same set. Ending with characters not specified is an error.

`exactly`(*quantity*, *setname*)
Spec method: require exactly this many characters from the set.

`length`(*length*)
Spec method: adjust the total length of passwords to generate.

`of`()
Commence a method chain building a fresh generator instance.

The generator instanciated by this call is new, but the character sets fed to it are always the same (the ones configured during `__init__()`). If a different character set is desired, a new instance of *`PasswordGenerator`* is neccessary.

The generator will be finalized by a *`done()`* call, and then used by calling *`generate()`* on it.

`some`(*setname*)
Spec method: use characters from the set, if they fit.

Once the you have specified your password scheme, you will have access to the generator instance.

class `grampg.`**`Generator`**(*sets*)
The generator object.

A generator instance undergoes three phases during its existance: create it with the character sets to choose from, specify it by calling its methods finalizing in a call to `done()`, and generate passwords with it by calling its *`generate()`* method.

Character sets should not be modified once the generator is instantiated. If other character sets are required, a new instance should be used.

During the specification, repeated calls to the same method (consecutively or otherwise) overrides previous calls, so it is not an error to call them more than once. Specification is over after a call `done()` succeds. Once *done*, the generator cannot be further spec'ed, and only calls to *`generate()`* are valid (although it is possible to call `done()` over and over again, it does not have effect).

Any attempt to add new specs to a *done* generator will raise *`PasswordGeneratorIsDone`*.

Note: Generator instances should be built by means of *`PasswordGenerator`*, and only the *`generate()`* method should ever be directly called on instances of this class.

`generate`()
Return one generated password based on the collected specs.

Can be called any number of times, each yielding a new, independant password.

Raises *`PasswordSpecsNonValidatedError`* if the generator is not *done* (the `done()` method has not yet been called). Raises `PasswordSpecError` if frame spec methods (`length`, `beginning_with` `ending_with`) collide.

In case of errors during the specification, the following exceptions are used.

exception `grampg.`**`PasswordSpecsError`**
Root of grampg exceptions.

Itself used to signal errors during specification or validation of a generator.

exception `grampg.`**`PasswordSpecsNonValidatedError`**
Raised when *`generate()`* is called on a generator before a it is *done*.

exception `grampg.`**`PasswordGeneratorIsDone`**

Raised when a new specification is attempted on a *done* generator.

CHAPTER 3

The password generation algorithm

XXX describe the algorithm, its strengh and limitation.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index