coolname Documentation

Release 1.1.0

Alexander Lukanin

Aug 02, 2018

Contents

CHAPTER 1

Configuration rules

Configuration is a flat dictionary of rules:

```
{
    '<rule_id>': {
        'comment': 'Some info about this rule. Not mandatory.',
        'type': '<nested|cartesian|words|phrases|const>',
        # additional fields, depending on type
    },
    ...
}
```

`<rule_id>` is the identifier of rule. Root rule must be named `'all'` - that's what you use when you call *`generate()`* or *`generate_slug()`* without arguments.

There are five types of configuration rules.

1.1 Words list

A ground-level building block. Chooses a random word from a list, with equal probability.

```
# This will produce random color
'color': {
    'type': 'words',
    'words': ['red', 'green', 'yellow']
},
# This will produce random taste
'taste': {
    'type': 'words',
    'words': ['sweet', 'sour']
},
# This will produce random fruit
'fruit': {
```

(continues on next page)

(continued from previous page)

```
    'type': 'words',
    'words': ['apple', 'banana']
},
```

1.2 Phrases list

Same as words list, but each element is one or more words.

```
# This will produce random color
'color': {
    'type': 'phrases',
    'words': ['red', 'green', 'navy blue', ['royal', 'purple']]
}
```

Phrase can be written as a string (words are separated by space) or as a list of words.

1.3 Nested list

Chooses a random word (or phrase) from any of the child lists. Probability is proportional to child list length.

```
# This will produce random adjective: color or taste
'adjective': {
    'type': 'nested',
    'lists': ['color', 'taste']
},
```

Child lists can be of any type.

Number of child lists is not limited.

Length of nested list is the sum of lengths of all child lists.

1.4 Constant

It's just a word. Useful for prepositions.

```
'of': {
    'type': 'const',
    'value': 'of'
},
```

1.5 Cartesian list

Cartesian list works like a slot machine, and produces a list of length N by choosing one random word (or phrase) from every child list.

```
# This will produce a random list of 4 words,
# for example: ['my', 'banana', 'is', 'sweet']
'all': {
    'type': 'cartesian',
    'lists': ['my', 'fruit', 'is', 'adjective']
},
# Additional const definitions
'is': {
    'type': 'const',
    'value': 'is'
},
'my': {
    'type': 'const',
    'value': 'my'
},
```

Length of Cartesian list is the product of lengths of child lists.

Let's try the config defined above:

```
>>> from coolname import RandomGenerator
>>> generator = RandomGenerator(config)
>>> for i in range(3):
...     print(generator.generate_slug())
...
my-banana-is-sweet
my-apple-is-green
my-apple-is-sour
```

Warning: You can have many nested lists, but you should never put a Cartesian list inside another Cartesian list.

CHAPTER 2

Length limits

2.1 Number of characters

There are two limits:

- `max_length`

  This constraint is hard: you can't create *`RandomGenerator`* instance if some word (or phrase) in some rule exceeds that rule's limit.

  For example, this will fail:

  ```
  {
      "all": {
          "type": "words",
          "words": ["cat", "tiger", "jaguar"],
          "max_length": 5
      }
  }
  ```

  Different word lists and phrase lists can have different limits. If you don't specify it, there is no limit.

  Note: when max_length is applied to phrase lists, spaces are not counted. So this will work:

  ```
  {
      "all": {
          "type": "phrases",
          "phrases": ["big cat"],
          "max_length": 6
      }
  }
  ```

- `max_slug_length`

  This constraint is soft: if result is too long, it is silently discarded and generator rolls the dice again. This allows you to have longer-than-average words (and phrases) which still fit nicely with shorter

words (and phrases) from other lists.

Of course, it's better to keep the fraction of "too long" combinations low, as it affects the performance. In fact, *RandomGenerator* performs a sanity test upon initialization: if probability of getting "too long" combination is unacceptable, it will raise an exception.

For example, this will produce 7 possible combinations, and 2 combinations (green-square and green-circle) will never appear because they exceed the max slug length:

```
{
    "adjective": {
        "type": "words",
        "words": ["red", "blue", "green"]
    },
    "noun": {
        "type": "words",
        "words": ["line", "square", "circle"]
    },
    "all": {
        "type": "cartesian",
        "lists": ["adjective", "noun"],
        "max_slug_length": 11
    }
}
```

Both of these limits are optional. Default configuration uses `max_slug_length = 50` according to Django slug length.

2.2 Number of words

Use `number_of_words` parameter to enforce particular number of words in a phrase for a given list.

This constraint is hard: you can't create *RandomGenerator* instance if some phrase in a given list has a wrong number of words.

For example, this will fail because the last item has 3 words:

```
{
    "all": {
        "type": "phrases",
        "phrases": [
            "washing machine",
            "microwave oven",
            "vacuum cleaner",
            "large hadron collider"
        ],
        "number_of_words": 2
    }
}
```

CHAPTER 3

Configuration files

Another small example: a pair of (adjective, noun) generated as follows:

```
(crouching|hidden) (tiger|dragon)
```

Of course, you can just feed config dict into *RandomGenerator* constructor:

```
>>> from coolname import RandomGenerator
>>> config = {'all': {'type': 'cartesian', 'lists': ['adjective', 'noun']}, 'adjective
↪': {'type':'words', 'words':['crouching','hidden']}, 'noun': {'type': 'words',
↪'words': ['tiger', 'dragon']}}
>>> g = RandomGenerator(config)
>>> g.generate_slug()
'hidden-dragon'
```

but it becomes inconvenient as number of words grows. So, *coolname* can also use a mixed files format: you can specify rules in JSON file, and encapsulate long word lists into separate plain txt files (one file per one `"words"` rule).

For our example, we would need three files in a directory:

my_config/config.json

```
{
    "all": {
        "type": "cartesian",
        "lists": ["adjective", "noun"]
    }
}
```

my_config/adjective.txt

```
crouching
hidden
```

my_config/noun.txt

```
dragon
tiger
```

Note: only config.json is mandatory; you can name other files as you want.

Use auxiliary function to load config from a directory:

```
>>> from coolname.loader import load_config
>>> config = load_config('./my_config')
```

That's all! Now loaded config contains all the same rules and we can create *RandomGenerator* object:

```
>>> config
{'adjective': {'words': ['crouching', 'hidden'], 'type': 'words'}, 'noun': {'words': [
↪'dragon', 'tiger'], 'type': 'words'}, 'all': {'lists': ['adjective', 'noun'], 'type
↪': 'cartesian'}}
>>> g = RandomGenerator(config)
>>> g.generate_slug()
'hidden-tiger'
```

3.1 Text file format for words

Basic format is simple:

```
# comment
one
two  # inline comment

# blank lines are OK
three
```

You can also specify options like this:

```
max_length = 13
```

Which is equivalent to adding the same option in config dictionary:

```
{
    'type': 'words',
    'words': ['one', 'two', 'three'],
    'max_length': 13
}
```

Options should be placed in the beginning of the text file, before the first word.

3.2 Text file format for phrases

For phrases, format is the same as for words. If any line in a file has more than one word, the whole file is automagically transformed to a `"phrases"` list instead of `"words"`.

For example, this file:

```
one
two

# Here is the phrase
three four
```

is translated to the following rule:

```
{
    "type": "phrases",
    "phrases": [
        ["one"], ["two"], ["three", "four"]
    ]
}
```

CHAPTER 4

Unicode support

Default implementation uses English, but you can create configuration in any language - just save the config files in UTF-8 encoding.

CHAPTER 5

Environment variables

You can replace the default generator using one or both following variables:

```
export COOLNAME_DATA_DIR=some/path
export COOLNAME_DATA_MODULE=some.module
```

If *any* of these is set and not empty, default generator is not created (saving memory), and your custom generator is used instead.

5.1 COOLNAME_DATA_DIR

It must be a valid path (absolute or relative) to the directory with `config.json` and `*.txt` files.

5.2 COOLNAME_DATA_MODULE

It must be a valid module name, importable from the current Python environment.

It must contain a variable named `config`, which is a dictionary (see *Configuration rules*).

Adjust `sys.path` (or `PYTHONPATH`) if your module fails to import.

5.3 Precedence

1. If `COOLNAME_DATA_DIR` is defined and not empty, *and the directory exists*, it is used.
2. If `COOLNAME_DATA_MODULE` is defined and not empty, it is used.
3. Otherwise, `ImportError` is raised.

The reason for this order is to support packaging in egg files. If you don't care about eggs, use only `COOLNAME_DATA_DIR` because it's more efficient and easier to maintain.

CHAPTER 6

Randomization

6.1 Re-seeding

As a source of randomness *coolname* uses standard `random` module, specifically `random.randrange()` function.

To re-seed the default generator, simply call `random.seed()`:

```
import os, random
random.seed(os.urandom(128))
```

coolname itself never calls `random.seed()`.

6.2 Replacing the random number generator

By default, all instances of *RandomGenerator* share the same random number generator.

To replace it for a custom generator:

```
from coolname import RandomGenerator
import random, os
seed = os.urandom(128)
generator = RandomGenerator(config, random=random.Random(seed))
```

To replace it for *coolname.generate()* and *coolname.generate_slug()*:

```
import coolname
import random, os
seed = os.urandom(128)
coolname.replace_random(random.Random(seed))
```

6.3 How randomization works

In this section we dive into details of how *coolname* generates random sequences.

Let's say we have following config:

```
config = {
    'all': {
        'type': 'cartesian',
        'lists': ['price', 'color', 'object']
    },
    # 2 items
    'price': {
        'type': 'words',
        'words': ['cheap', 'expensive']
    },
    # 3 items
    'color': {
        'type': 'words',
        'words': ['black', 'white', 'red']
    },
    # 5 + 6 = 11 items
    'object': {
        'type': 'nested',
        'lists': ['footwear', 'hat']
    },
    # 5 items
    'footwear': {
        'type': 'words',
        'words': ['shoes', 'boots', 'sandals', 'sneakers', 'socks']
    },
    # 6 items
    'hat': {
        'type': 'phrases',
        'phrases': ['top hat', 'fedora', 'beret', 'cricket cap', 'panama', 'sombrero']
    }
}
import coolname
generator = coolname.RandomGenerator(config)
```

The overall number of combinations is $2 \times 3 \times (5 + 6) = 66$.

You can imagine a space of possible combinations as a virtual N-dimensional array. In this example, it's 3-dimensional, with sides equal to 2, 3 and 11.

When user calls *RandomGenerator.generate_slug()*, a random integer is generated via `randrange(66)`. Then, the integer is used to pick an element from 3-dimensional array.

Table 1: Possible combinations

`randrange()` returns	*`generate_slug()`* returns
0	cheap-black-top-hat
1	cheap-black-fedora
2	cheap-black-beret
3	cheap-black-cricket-cap
4	cheap-black-panama
5	cheap-black-sombrero
6	cheap-black-shoes
7	cheap-black-boots
8	cheap-black-sandals
9	cheap-black-sneakers
10	cheap-black-socks
11	cheap-white-top-hat
12	cheap-white-fedora
...	...
63	expensive-red-sandals
64	expensive-red-sneakers
65	expensive-red-socks

Note: Actual order of combinations is an implementation detail, you should not rely on it.

CHAPTER 7

Classes and functions

Functions and methods accept and return Unicode strings, that is, `unicode` in Python 2 and `str` in Python 3.

7.1 Default generator

coolname.**generate**(*pattern=None*)
Returns a random sequence as a list of strings.

Parameters **pattern** (*int*) – Can be 2, 3 or 4.

Return type list of strings

coolname.**generate_slug**(*pattern=None*)
Same as *generate()*, but returns a slug as a string.

Parameters **pattern** (*int*) – Can be 2, 3 or 4.

Return type str

coolname.**get_combinations_count**(*pattern=None*)
Returns the number of possible combinations.

Parameters **pattern** (*int*) – Can be 2, 3 or 4.

Return type int

coolname.**replace_random**(*random*)
Replaces the random number generator. It doesn't affect custom generators.

Parameters **random** – `random.Random` instance.

7.2 Custom generators

class coolname.**RandomGenerator**(*config*, *random=None*)

Parameters

- **config** (*dict*) – Custom configuration dictionary.
- **random** – `random.Random` instance. If not provided, `random.randrange()` will be used.

generate(*pattern=None*)
Returns a random sequence as a list of strings.

Parameters **pattern** – Not applicable by default. Can be configured.

Return type list of strings

generate_slug(*pattern=None*)
Same as *generate()*, but returns a slug as a string.

Parameters **pattern** – Not applicable by default. Can be configured.

Return type str

get_combinations_count(*pattern=None*)
Returns the number of possible combinations.

Parameters **pattern** – Not applicable by default. Can be configured.

Return type int

CHAPTER 8

Is it thread-safe?

`coolname` is thread-safe and virtually stateless. The only shared state is the global `random.Random` instance, which is also thread-safe. You can re-seed or even completely override it, see *Randomization*.

CHAPTER 9

Release history

9.1 1.1.0 (2018-08-02)

- 32-bit Python is supported.

9.2 1.0.4 (2018-02-17)

- **Breaking changes:**
  - Renamed `RandomNameGenerator` to `RandomGenerator`.
  - `randomize()` was removed, because it was just an alias to `random.seed()`.
- Phrase lists give you even more freedom when creating custom generators.
- You can seed or even replace the underlying `random.Random` instance, see Randomization.
- Change the default generator using `COOLNAME_DATA_DIR` and `COOLNAME_DATA_MODULE`. This also saves memory!
- Total number of combinations = 60 billions.

9.3 0.2.0 (2016-09-28)

- More flexible configuration: `max_length` and `max_slug_length` constraints. See documentation.
- Total number of combinations increased from 43 to 51 billions.

9.4 0.1.1 (2015-12-17)

- Consistent behavior in Python 2/3: output is always unicode.

- Provide `from coolname.loader import load_config` as a public API for loading custom configuration.
- More strict configuration validation.
- Total number of combinations increased from 33 to 43 billions.

9.5 0.1.0 (2015-11-03)

- First release on PyPI.

CHAPTER 10

Random Name and Slug Generator

Do you want random human-readable strings?

```
>>> from coolname import generate_slug
>>> generate_slug()
'big-maize-lori-of-renovation'
>>> generate_slug()
'tunneling-amaranth-rhino-of-holiness'
>>> generate_slug()
'soft-cuddly-shrew-of-expertise'
```

10.1 Features

- Generate slugs, ready to use, Django-compatible.

  ```
  >>> from coolname import generate_slug
  >>> generate_slug()
  'qualified-agama-of-absolute-kindness'
  ```

- Generate names as sequences and do whatever you want with them.

  ```
  >>> from coolname import generate
  >>> generate()
  ['beneficial', 'bronze', 'bee', 'of', 'glee']
  >>> ' '.join(generate())
  'limber transparent toad of luck'
  >>> ''.join(x.capitalize() for x in generate())
  'CalmRefreshingTerrierOfAttraction'
  ```

- Generate names of specific length: 2, 3 or 4 words.

```
>>> generate_slug(2)
'mottled-crab'
>>> generate_slug(3)
'fantastic-acoustic-whale'
>>> generate_slug(4)
'military-diamond-tuatara-of-endeavor'
```

Note: without argument, it returns a random length, but probability of 4word name is much higher. Prepositions and articles (of, from, the) are not counted as words.

- Over 10^{10} random names.

Words	Combinations	Example
4	10^{10}	`talented-enigmatic-bee-of-hurricane`
3	10^{8}	`ambitious-turaco-of-joviality`
2	10^{5}	`prudent-armadillo`

```
>>> from coolname import get_combinations_count
>>> get_combinations_count(4)
60610181372
```

- Hand-picked vocabulary. `sexy` and `demonic` are about the most "offensive" words here - but there is only a pinch of them, for spice. Most words are either neutral, such as `red`, or positive, such as `brave`. And subject is always some animal, bird, fish, or insect - you can't be more neutral than Mother Nature.
- Easy customization. Create your own rules!

```
>>> from coolname import RandomGenerator
>>> generator = RandomGenerator({
...   'all': {
...     'type': 'cartesian',
...     'lists': ['first_name', 'last_name']
...   },
...   'first_name': {
...     'type': 'words',
...     'words': ['james', 'john']
...   },
...   'last_name': {
...     'type': 'words',
...     'words': ['smith', 'brown']
...   }
... })
>>> generator.generate_slug()
'james-brown'
```

10.2 Installation

```
pip install coolname
```

coolname is written in pure Python and has no dependencies. It works on any modern Python version, including PyPy.

Python Module Index

C

Index