StringGenerator

Release 0.4.0

Paul Wolf

Mar 20, 2021

CONTENTS:

Generate test data, unique ids, passwords, vouchers or other randomized textual data very quickly using a template language. The template language is superficially similar to regular expressions but instead of defining how to match or capture strings, it defines how to generate randomized strings. A very simple invocation to produce a random string with word characters of 30 characters length:

```
from strgen import StringGenerator as SG
SG(r"[\w]{30}").render()
'wQjLVRIj1sjjslORpqLJyDObaCnDR2'
```

Generate 50000 unique secure tokens, each 32 characters:

```
secure_tokens = SG("[\p\w]{32}").render_set(50000)
```

Install:

```
pip install StringGenerator
```

The current package requires Python 3.6 or higher. Use version 0.3.4 or earlier if you want to use Python 2.7 or an earlier Python 3 version.

The purpose of this module is to save the Python developer from having to write verbose code around the same pattern every time to generate passwords, keys, tokens, test data, etc. of this sort:

```
my_secret_key = ''.join(random.choice(string.ascii_uppercase + string.digits) for x␣
↪in range(30))
```

that is:

1. Hard to read even at this simplistic level.
2. Hard to safely change quickly. Even modest additions to the requirements need unreasonably verbose solutions.
3. Doesn't use safe encryption standards.
4. Doesn't provide the implied minimal guarantees of character occurance.
5. Hard to track back to requirements ("must be between x and y in length and have characters from sets Q, R and S").

The template uses short forms similar to those of regular expressions. An example template for generating the same secret key as above:

```
SG(r"[\w\p]{30}").render()
```

will generate something like the following:

```
"\\/]U.`I$9E[#!'HTT;MSH].-Y};C|Y"
```

Changing the character specification is a matter of adding two characters.

Guarantee at least two "special" characters in a string:

```
[\w\p]{10}&[\p]{2}
```

You can also generate useful test data, like fake emails with plenty of variation:

```
[\c]{10}.[\c]{5:10}@[\c]{3:12}.(com|net|org)
```

CHAPTER

ONE

TESTING AND DEVELOPING

For running the unit tests or making changes, you might want to try:

```
python -m venv .venv && source .venv/bin/activate && pip install --upgrade pip && pip␣
↪install -r requirements.txt
pytest
```

Hypothesis is used in some unit tests

CHAPTER

TWO

LICENSE

Released under the BSD license.

CHAPTER

THREE

ACKNOWLEDGEMENTS

Thanks to Robert LeBlanc who caught some important errors in escaping special characters. Thanks to Andreas Motl for the progress counter.

Original Author: paul.wolf@yewleaf.com

3.1 Installation

From version 0.4.0, support for Python 2.7 is dropped. If you still need support for Python 2, use version StringGenerator 0.3.4.

There are no dependencies beyond the Python Standard Library.

Install as standard for Python packages from PyPi:

```
pip install StringGenerator
```

3.2 Usage

Throughout we import the StringGenerator class aliased as SG:

```
from strgen import StringGenerator as SG
```

Generate a unique string using this syntax:

```
SG(<template>).render()
```

or to produce a list of unique strings:

```
SG(<template>).render_set(<length>)
```

- template: a string that conforms to the StringGenerator pattern language as defined in this document
- length: the list of the result set. *render_set()* produces a unique set. *render_list()* can be used if uniqueness is not required.

Example:

```
SG('[\l\d]{4:18}&[\d]&[\p]').render()
'Cde90uC{X6lWbOueT'
```

The `template` is a string that is a sequence of one or more of the following:

- *Literal text* (for example: `UID`)
- *Character class* (for example: `[a-z\s]`)
- *Group*, a combination of literals and character classes, possibly separated by operators and using parentheses where appropriate (for example: `(UID[\d]{4}&[\w]{4})`)

A quantifier `{x-y}` should be provided normally just after a character class to determine the size of the resulting string where x and y are integers and y can be left away for a fixed length. With no quantifier, a character class assumes a length of 1.

The range operator can be either `-` or `:` as in `{10:20}` or `{10-20}`.

You can avoid escaping if you use raw strings, like `r"[\u]{20}"`. You need to accommodate f-strings by doubling the curly braces for quantifiers. You can have raw f-strings:

```
x = "foo"
SG(fr"[\d\u]{{1:20}}{x}").render()
'E3ZG2foo'
```

If you want consistent results for testing, you can seed the generator:

```
SG("[\w]{20}", seed=1234).render_set(10000)
```

seed is any integer as per random if you would like to test with reproduceable results.

See *Seeding and Custom Randomization*

3.2.1 Literal: <any string>

Any literal string.

Example:

```
orderno
```

Special characters need to be escaped with backslash `\`.

3.2.2 Character class: [<class specification>]

Much like in regular expressions, it uses strings of characters and hyphen for defining a class of characters.

Example:

```
[a-zA-Z0-9_]
```

The generator will randomly choose characters from the set of lower case letters, digits and the underscore. The number of characters generated will be exactly one in this case. For more, use a quantifier:

```
[a-zA-Z0-9_]{8}
```

As a shortcut for commonly used character sets, a character set code may be used. The following will render in exactly the same way:

```
[\w]{8}
```

3.2.3 Character Set Codes

- \W: whitespace + punctuation
- \a: ascii_letters
- \c: lowercase
- \d: digits
- \h: hexdigits
- \l: letters
- \o: octdigits
- \p: punctuation
- \r: printable
- \s: whitespace
- \u: uppercase
- \U: uppercase
- \w: _ + letters + digits

Escape \u as \\u since this is the unicode prefix unless you use a raw string:

```
r"[\u]{8}"
```

Or use the alias \U.

3.2.4 Quantifier: {x:y}

Where x is lower bound and y is upper bound. This construct must always follow immediately a class with no intervening whitespace. It is possible to write {:y} as a shorthand for {0:y} or {y} to indicate a fixed length. {x-y} and {x:y} are synonymous.

Example:

```
[a-z]{0:8}
```

Generates a string from zero to 8 in length composed of lower case alphabetic characters.

```
[a-z]{4}|[0-9]{4}
```

Generates a string with either four lower case alphabetic characters or a string of digits that is four in length.

Using a character class and no quantifier will result in a quantifier of 1. Thus:

```
[abc]
```

will result always in either a, b, or c.

3.2.5 Data Sources

We provide the `${varname}` syntax to enable any value to be returned. `varname` must be provided as a keyword argument to the `render()`, `render_set()` or `render_list()` methods. You can use a list, function (callable) or generator. Here's an example using a list:

```
SG('William of ${names}').render(names=['Orange', 'Normandy', 'Ockham'])
```

Or use a range converted to a list:

```
SG('You have ${chances} chances').render(chances=list(range(1000)))
```

Or using a function:

```
SG('William of ${names}').render(names=lambda: random.choice(['Orange', 'Normandy',
↪'Ockham']))
```

You can obviously pass any callable or generator that might, for instance, randomly choose a value from a database, if that is what you want.

Note there is a difference in handling between a callable and list type. If you use a `list`, StringGenerator picks an item from the list for you, randomly. If you use a callable, StringGenerator takes and inserts whatever is returned by the callable. The callable is required to do any randomization if that is what the user wants. So, if you pass a function that returns a list, the entire list will be inserted as a string.

As mentioned above, if you use an f-string, double your curly braces for the data source name.

```
x = "William of "
SG(f"{x}${{names}}").render(names=['Orange', 'Normandy', 'Ockham'])
```

3.2.6 Group: (<group specification>)

A group specification is a collection of literals, character classes or other groups divided by the OR operator `|` or the shuffle operator `&`.

3.2.7 OR Operator

The binary `|` operator can be used in a group to cause one of the operands to be returned and the other to be ignored with an even chance.

3.2.8 Shuffle Operator

The binary `&` operator causes its operands to be combined and shuffled. This addresses the use case for many password requirements, such as, "at least 6 characters where 2 or more are digits". For instance:

```
[\l]{6:10}&[\d]{2}
```

If a literal or a group is an operand of the shuffle operator, it will have its character sequence shuffled with the other operand.

```
foo&bar
```

will produce strings like:

```
orbfao
```

3.2.9 Concatenation and Operators

Classes, literals and groups in sequence are concatenated in the order they occur. Use of the | or & operators always binds the operands immediately to the left and right:

```
[\d]{8}xxx&yyy
```

produces something like:

```
00488926xyyxxy
```

In other words, the digits occur first in sequence as expected. This is equivalent to this:

```
[\d]{8}(xxx&yyy)
```

3.2.10 Special Characters, Escaping and Errors

There are fewer special characters than regular expressions:

```
[]{}()|&$\-
```

They can be used as literals by escaping with backslash. All other characters are treated as literals. The hyphen is only special in a character class, when it appears within square brackets.

One special case is the escape character itself, backslash \. To escape this, you will need at least two backslashes. So, three altogether: one for Python's string interpretation and one for StringGenerator's escaping. If for some exotic reason you want two literal backslashes in a row, you need a total of eight backslashes. The foregoing presupposes the template is a string in a file. If you are using the template in a shell command line or shell script, you'll need to make any changes required by your specific shell.

The template parser tries to raise exceptions when syntax errors are made, but not every error will be caught, like having space between a class and quantifier.

3.2.11 Spaces

Do not use any spaces in the template unless you intend to use them as characters in the output:

```
>>> SG('(zzz & yyy)').render()
u'zzyz y y'
```

3.2.12 Character Classes and Quantifiers

Use a colon in the curly braces to indicate a range. There are sensible defaults:

```
[\w]       # randomly choose a single word character
[\w]{0:8}  # generate word characters from 0-8 in length
[\w]{:8}   # a synonym for the above
[\w]{8}    # generate word characters of exactly 8 in length
[a-z0-9]   # generate a-z and digits, just one as there is no quantifier
[a-z0-9_!@]  # you can combine ranges with individual characters
```

As of version 0.1.7, quantifier ranges can alternatively be specified with a hyphen:

```
[\w]{4-8}
```

Here's an example of generating a syntactically valid but, hopefully, spurious email address:

```
[\c]{10}(.|_)[\c]{5:10}@[\c]{3:12}.(com|net|org)
```

The first name will be exactly 10 lower case characters; the last name will be 5-10 characters of lower case letters, each separated by either a dot or underscore. The domain name without domain class will be 3 - 12 lower case characters and the domain type will be one of '.com','.net','.org'.

The following will produce strings that tend to have more letters, because the set of letters (52) is larger than the set of digits (10):

```
[\l\d]
```

Using multiple character set codes repeatedly will increase the probability of a character from that set occuring in the result string:

```
[\l\d\d\d\d]
```

This will provide a string that is three times more likely to contain a digit than the previous example.

3.2.13 Uniqueness

`render_list()` and `render_set()` both produce unqiue sequences of strings. In general, you should use `render_set()` as it's much faster. See *render_set() vs. render_list()*.

When using the `unique=True` flag in the `render_list()` method, it's possible the generator cannot possibly produce the required number of unique strings. For instance:

```
SG("[0-1]").render_list(100, unique=True)
```

This will generate an exception but not before attempting to generate the strings.

The number of times the generator needs to render new strings to satisfy the list length and uniqueness is not determined at parse time. The maximum number of times it will try is by default n x 10 where n is the requested length of the list. Therefore, taking the above example, the generator will attempt to generate the unique list of 0's and 1's 100 x 10 = 1000 times before giving up.

3.3 *render_set()* vs. *render_list()*

render_set(<set_size>) was introduced in StringGenerator 0.4.2. It is an alternative to *render_list()*.

render_list() takes an argument for a progress callback function and can generate a unique list if *unique=True*. It also throws an exception if you provide a pattern and result size that are not compatible, like:

```
SG("[123456789]{3}").render_list(800, unique=True)
```

The maximum number of unique results for this pattern is 729. An exception will be raised:

```
UniquenessError: couldn't satisfy uniqueness
```

In contrast, *render_set()* returns a *set* so it does not need a *unique=True* parameter. It is optimised to be fast and therefore does not support either a progress callback nor will it check if the request is feasible.

But it's much faster than *render_list(count, unique=True)*

On a reasonably fast host, you can generate 100,000 unique 20 character strings in under five seconds:

```
In [5]: %time tokens = SG("[\w]{20}").render_set(100000)
CPU times: user 3.84 s, sys: 598 ms, total: 4.44 s
Wall time: 4.44 s
```

render_list(100000, unique=True) would accomplish the same thing but take well over 10x as long.

The *seed* parameter has the same effect with *render_set()*:

```
In [23]: SG(r"[\u\d]{46}", seed=100).render_set(10)
Out[23]:
{'1A2SM257I4JNSYABZHPJP4L0TS7YYY4H9IC6DB61YYOBHE',
'6SKXTOZ1CARF8GR148R9C5TLFF5N7FVJZ6CZV0L1PF2XAS',
'9DP6SV7V8X4ZDGT1QP165QJDUKDRHDOT7NWV3AH4M7HNCY',
'GZJ48C2KD4DMPKGCINFHQEZC8142V37MQ0JN0ERZQGMODY',
'H2BN7K4C82WAF5D1YJM1LNRQNSZZQVBRP7LD0FWHK4XN7P',
'HTIHODZ31WFEC20BJ4GFSG9JI9JKWYG5UH4SOWOCAO6H9F',
'I4D4W64N5D6QB77GLCO3AHOXUHTGTT2MUFKVJM1FFUNURJ',
'J33LZW16H8HF3QDNVOTNLJMWX0NZ39RYKHLAZJKMKBP2YI',
'RARTCB9SOZAFOTBFWU2CLFBW9JKAAU6ZLYY2MNK9RE3I16',
'ZB3VCQ5DEWCQ3S8ZXUKILV5Z0P5G7NLEPFBR4OV0CR4WVE'}

In [24]: SG(r"[\u\d]{46}", seed=100).render_set(10)
Out[24]:
{'1A2SM257I4JNSYABZHPJP4L0TS7YYY4H9IC6DB61YYOBHE',
'6SKXTOZ1CARF8GR148R9C5TLFF5N7FVJZ6CZV0L1PF2XAS',
'9DP6SV7V8X4ZDGT1QP165QJDUKDRHDOT7NWV3AH4M7HNCY',
'GZJ48C2KD4DMPKGCINFHQEZC8142V37MQ0JN0ERZQGMODY',
'H2BN7K4C82WAF5D1YJM1LNRQNSZZQVBRP7LD0FWHK4XN7P',
'HTIHODZ31WFEC20BJ4GFSG9JI9JKWYG5UH4SOWOCAO6H9F',
'I4D4W64N5D6QB77GLCO3AHOXUHTGTT2MUFKVJM1FFUNURJ',
'J33LZW16H8HF3QDNVOTNLJMWX0NZ39RYKHLAZJKMKBP2YI',
'RARTCB9SOZAFOTBFWU2CLFBW9JKAAU6ZLYY2MNK9RE3I16',
'ZB3VCQ5DEWCQ3S8ZXUKILV5Z0P5G7NLEPFBR4OV0CR4WVE'}
```

3.4 Seeding and Custom Randomization

3.4.1 Seeding

If you want to test and produce consistent results, you can use the `seed` option:

```
In [79]: SG("[\w]{10}&[\d]{10}", seed=4318).render_list(10)
Out[79]:
['42U1uqGt5qG0519J6562',
 'q5d68X3n66r5h81pdz59',
 '1u307GH7Ad9Xmd1Zg119',
 'I7R224u5efft78A6986h',
 '3TP43404okw7Q5R019l4',
 '0D3952i05QYe1C935y36',
 '219W9iw9924XYy368E4B',
 '5F74c59Sy947LNw28p86',
 'W3ocGxWw675166352862',
 '35A6G69rEjh4U58t9X5E']

In [80]: SG("[\w]{10}&[\d]{10}", seed=4318).render_list(10)
Out[80]:
['42U1uqGt5qG0519J6562',
 'q5d68X3n66r5h81pdz59',
 '1u307GH7Ad9Xmd1Zg119',
 'I7R224u5efft78A6986h',
 '3TP43404okw7Q5R019l4',
 '0D3952i05QYe1C935y36',
 '219W9iw9924XYy368E4B',
 '5F74c59Sy947LNw28p86',
 'W3ocGxWw675166352862',
 '35A6G69rEjh4U58t9X5E']
```

Notice these two lists are exactly the same.

Normally, when the StringGenerator is initialised, it will try to use `random.SystemRandom` as the random method provider. It falls back gracefully to `random.Random` if that is not available.

When you use the `seed` option, it will force use of `random.Random` and use the provided seed value, which can be any integer. This will cause the results to be the same each time you initialise the StringGenerator.

3.4.2 Custom Random Class

You can also provide your own Random class. Currently we use these methods:

- `choice()`
- `randint()`
- `shuffle()`

So, you'd need to provide at least these with the same arguments and return types as `Random` and `SystemRandom`.

See documentation for the Python Standard Library random package

> BEWARE: you should provide all the methods of `` `Random()` `` in your custom class because we might change the implementation of StringGenerator to use different methods.

3.5 Recipes

3.5.1 Produce Collisions

Quick check if two random sets might overlap:

```
In [121]: SG(r'[\u\d]{5}').render_set(10000) & SG(r'[\u\d]{5}').render_set(10000)
Out[121]: {'PH8AX'}
```

In this case, we had a single overlap. Easy to make that more overlapping by increasing set size with the same pattern:

```
In [59]: len(SG(r'[\u\d]{5}').render_set(100000) & SG(r'[\u\d]{5}').render_
↪set(100000))
Out[58]: 150
```

150 overlaps.

Or make it very unlikely that an overlap will exist by making the results more characters:

```
In [35]: len(SG(r'[\u\d]{50}').render_set(100000) & SG(r'[\u\d]{50}').render_
↪set(100000))
Out[35]: 0
```

It would be nice to know what the count of the sets of potential results are:

```
In [19]: SG(r'[\u\d]{5}').count()
Out[19]: 60466176

In [18]: SG(r'[\u\d]{50}').count()
Out[18]:␣
↪653318623500070906096690267158057820537143710472954871543071966369497141477376
```

The set `\u\d`:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789
```

is 36 characters, so `36^5 = 60466176`, permutation with replacement. And likewise `36^50 = a big number`.

3.5.2 Coin Flip

Flip a coin 10000 times and count heads and tails:

```
import collections
In [64]: collections.Counter(SG("heads|tails").render_list(10000))
Out[64]: Counter({'tails': 5046, 'heads': 4954})
```

Flip two coins and check how the combinations were distributed:

```
In [81]: collections.Counter(list(zip(SG("heads|tails").render_list(100), SG(
↪"heads|tails").render_list(100))))
Out[81]:
Counter({('heads', 'heads'): 20,
('tails', 'heads'): 21,
('heads', 'tails'): 25,
('tails', 'tails'): 34})
```

Roll two die 1000 times:

```
In [7]: d = list(zip(SG(r'[123456]').render_list(1000), SG(r'[123456]').render_
↪list(1000)))

In [8]: Counter(d)
Out[8]:
Counter({('2', '2'): 28,
         ('3', '1'): 28,
         ('4', '1'): 27,
         ('5', '2'): 32,
         ('1', '5'): 28,
         ('6', '2'): 42,
         ('4', '3'): 19,
         ('1', '4'): 40,
         ...
```

We counted where order matters, 36 potential outcomes.

3.5.3 Pick a card from a deck

First setup some definitions:

```
SPADE = ""
HEART = ""
DIAMOND = ""
CLUB = ""

def is_face_card(s):
   return s[0] in "JQK"

def is_black_suit(s):
   return s[-1] in (SPADE, CLUB)

def is_red_suit(s):
   return s[-1] in (DIAMOND, HEART)
```

Now let's randomly pick a card 1000 times:

```
In [208]: d = SG(f"J|Q|K|A|2|3|4|5|6|7|8|9|10[{HEART}{SPADE}{DIAMOND}{CLUB}]").render_
↪list(1000)

collections.Counter([is_face_card(x) & is_black_suit(x) for x in d])

In [209]: collections.Counter([is_face_card(x) & is_black_suit(x) for x in d])
Out[209]: Counter({False: 887, True: 113})
```

We expect the probability of getting a black face card to be 11.5%:

```
In [210]: 113/1000
Out[210]: 0.113
```

Close enough

3.5.4 Normal Distribution

We expect the distribution of picking one of 0 - 9 digits to be normally distributed if we try 100 times over 1000 times.

```
In [11]: import statistics

In [12]: d = [statistics.mean([int(SG("0|1|2|3|4|5|6|7|8|9").render()) for _ in␣
↪range(100)]) for _ in range(1000)]
   ...: nd = statistics.NormalDist.from_samples(d)
   ...: nd.stdev
Out[12]: 0.29718859089567784

In [13]: nd.mean
Out[13]: 4.51675
```

4.5 is the mean for 0 - 9.

3.6 Syntactical Details

The grammar for strgen is quite tolerant. We try to produce something unless there is confusion about the count of potential outcomes.

The shuffle operator *&* causes its operands to produce permutations without replacement.

While the shuffle operator is most useful as a binary operator, it can be used as a unary operator. These all do the same thing:

```
SG('123456789&').render()
SG('&123456789').render()
SG('&123456789&').render()
```

Likewise, the binary operator, |, no longer raises an exception if there is only one operand:

```
SG("1|")
```

as well as an extra operator:

```
SG("1|2|3|")
```

The associative properties should be observed. The following produces 2-character elements as output:

```
In [13]: SG("1|2|3[abc]").render_list(10)
Out[13]: ['1a', '3c', '3c', '1c', '2a', '3c', '3b', '3a', '1b', '2b']
```

This produces either one of : *1, 2, 3, a, b, c*.

```
In [14]: SG("1|2|3|[abc]").render_list(10)
Out[14]: ['1', '2', '1', 'a', 'a', '1', '2', '2', '1', '1']
```

and without square brackets:

```
In [138]: SG("1|2|3|abc").render_list(10)
Out[138]: ['1', '2', '3', 'abc', '1', '1', '2', 'abc', 'abc', '1']
```

3.7 StringGenerator.count()

> Note this feature is experimental and not fully tested at this time. It will probably not work for more complicated patterns.

`count()` is a somewhat experimental feature that shows the potential unique outcomes for a template pattern.

How many unique strings can we generate that are five characters long using upper case ascii and digits:

```
In [18]: SG(r'[\u\d]{5}').count()
Out[18]: 60466176
```

Because:

```
In [17]: len("ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789") ** 5
Out[17]: 60466176
```

Another example:

```
In [116]: SG(r'[abc]{1:3}|[\d]{2}|[\l]{3}').count()
Out[116]: 140747

In [117]: (3**1 + 3**2 + 3**3) + 10**2 + 52**3
Out[117]: 140747
```

This method calculates the potential unique results mathematically, but we can count every instance "manually" as well. First count how many we could get:

```
In [19]: SG(r'[abc]{5}').count()
Out[19]: 243
```

Produce every potential outcome:

```
In [28]: d = SG(r'[abc]{5}').render_list(243, unique=True)

In [29]: len(d)
Out[29]: 243
```

If we try to get 244 unique results, there will be an error:

```
In [22]: SG(r'[abc]{5}').render_list(244, unique=True)

---------------------------------------------------------------------------
...
UniquenessError: couldn't satisfy uniqueness
```

It does not mathematically calculate and then raise an error. It tries to produce the results and gives up after trying a certain number of times. You don't want to use `render_set()` here because it doesn't check if the result is not possible. It will never return.

One important thing to remember is each specified character is counted, even if it repeats another character in the sequence:

```
In [71]: SG("[xxxxxxxxxxx]{10}").count()
Out[71]: 61917364224
```

3.7.1 Limitations

`count()` will not know how to produce a result if you use a source variable that could be a callable or list. That's when using the `${somevariable}` syntax.

It will very specifically not work correct if you use the shuffle operator `&` on a complex template expression.

3.8 Progress Callback

When using the `progress_callback` parameter of the `render_list()` method, it's possible to inform others about the progress of string generation. This is especially useful when generating a large number of strings.

The callback function obtains two int parameters: `(current, total)`, which define the current progress and the total amount of requested strings.

By using that, callers of `render_list()` are able to implement a progress indicator suitable for informing end users about the progress of string generation.

3.9 Debugging

Call the dump() method on the class instance to get useful information:

- Version of strgen module
- Version of Python
- The class name used for random methods
- The parse tree
- The output from one invocation of the render() method

The output looks something like the following:

```
In [106]: SG('[\w]{8}&xyz|(zzz&yyy)').dump()
StringGenerator version: 0.4.0
Python version: 3.8.7 (default, Feb  3 2021, 06:31:03)
[Clang 12.0.0 (clang-1200.0.32.29)]
Random method provider class: SystemRandom
sequence:
OR
AND
     -1:8:_abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789
     xyz
sequence:
     AND
          zzz
          yyy
Out[106]: 'ybYxak7JRzN'
```

3.10 Rationale and Design Goals

In Python, the need to generate random strings comes up frequently and is accomplished usually (though not always) via something like the following code snippet:

```
import random
import string
mykey = ''.join(random.choice(string.ascii_uppercase + string.digits) for x in␣
↪range(10))
```

This generates a string that is 10 characters made of uppercase letters and digits. Unfortunately, this solution becomes cumbersome when real-world requirements are added. Take for example, the typical requirement to generate a password: "a password shall have 6 - 20 characters of which at least one must be a digit and at least one must be a special character". The above solution then becomes much more complicated and changing the requirements is an error-prone and unnecessarily complex task.

The equivalent using the strgen package:

```
from strgen import StringGenerator as SG
SG('[\u\d]{10}').render()
```

`strgen` is far more compact, flexible and feature-rich than using the standard solution:

- It tries to use a better entropy mechanism and falls back gracefully if this is not available on the host OS.
- The user can easily modify the specification (template) with minimal effort without the fear of introducing hard-to-test code paths.
- It covers a broader set of use cases: unique ids, persistent unique filenames, test data, etc.
- The template syntax is easy to learn for anyone familiar with regular expressions while being much simpler.
- It supports unicode.
- It works on Python 2.6, 2.7 and 3.x.
- It proposes a standard way of expressing common requirements, like "a password shall have 6 - 20 characters of which at least one must be a digit and at least one must be a special character":

  ```
  [\l\d]{4:18}&[\d]&[\p]
  ```

This package is designed with the following goals in mind:

- Provide an abstract template language that does not depend on a specific implementation language.
- Reduce dependencies on other packages.
- Keep syntax as simple as possible while being useful.
- Provide an implementation design with associated behaviour that strikes the right balance between ease-of-implementation and ease-of-use.
- Superficially similar to regular expressions to enable developers to quickly pick up the template syntax.
- Support non-ASCII languages (unicode).

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search