Python String Utils Documentation

Release 1.0.0

Davide Zanotti

Jul 16, 2026

CONTENTS

1 Installing **3**

2 Checking installed version **5**

3 Library structure **7**

4 Modules **9**
- 4.1 String Validation 9
- 4.2 String Manipulation 16
- 4.3 String Generation 21
- 4.4 Errors 23

5 Function Indices **25**

Python Module Index **27**

Index **29**

This is a handy library to validate, manipulate and generate strings, which is:

- Simple and "pythonic"
- Fully documented and with examples!
- 100% code coverage!
- Tested against all officially supported Python versions: 3.5, 3.6, 3.7, 3.8.
- Fast (mostly based on compiled regex)
- Free from external dependencies
- PEP8 compliant

CHAPTER

ONE

INSTALLING

```
>>> pip install python-string-utils
```

CHAPTER

TWO

CHECKING INSTALLED VERSION

```
>>> import string_utils
>>> string_utils.__version__
>>> '1.0.0' # (if '1.0.0' is the installed version)
```

CHAPTER

THREE

LIBRARY STRUCTURE

The library basically consists in the python package *string_utils*, containing the following modules:

- *validation.py* (contains string check api)
- *manipulation.py* (contains string transformation api)
- *generation.py* (contains string generation api)
- *errors.py* (contains library-specific errors)
- *_regex.py* (contains compiled regex **FOR INTERNAL USAGE ONLY**)

Plus a secondary package *tests* which includes several submodules. Specifically one for each test suite and named according to the api to test (eg. tests for *is_ip()* will be in *test_is_ip.py* and so on). All the public API are importable directly from the main package *string_utils*, so this:

```
>>> from string_utils.validation import is_ip
```

can be simplified as:

```
>>> from string_utils import is_ip
```

CHAPTER

FOUR

MODULES

4.1 String Validation

`string_utils.validation.`**`contains_html`**(*input_string: str*) → bool

Checks if the given string contains HTML/XML tags.

By design, this function matches ANY type of tag, so don't expect to use it as an HTML validator, its goal is to detect "malicious" or undesired tags in the text.

Examples:

```
>>> contains_html('my string is <strong>bold</strong>') # returns true
>>> contains_html('my string is not bold') # returns false
```

Parameters
input_string (*str*) – Text to check

Returns
True if string contains html, false otherwise.

`string_utils.validation.`**`is_camel_case`**(*input_string: Any*) → bool

Checks if a string is formatted as camel case.

A string is considered camel case when:

- it's composed only by letters ([a-zA-Z]) and optionally numbers ([0-9])
- it contains both lowercase and uppercase letters
- it does not start with a number

Examples:

```
>>> is_camel_case('MyString') # returns true
>>> is_camel_case('mystring') # returns false
```

Parameters
input_string (*str*) – String to test.

Returns
True for a camel case string, false otherwise.

`string_utils.validation.`**`is_credit_card`**(*input_string: Any*, *card_type: str = None*) → bool

Checks if a string is a valid credit card number. If card type is provided then it checks against that specific type only, otherwise any known credit card number will be accepted.

Supported card types are the following:

- VISA
- MASTERCARD
- AMERICAN_EXPRESS
- DINERS_CLUB
- DISCOVER
- JCB

Parameters

- **input_string** (*str*) – String to check.
- **card_type** (*str*) – Card type. Default to None (any card).

Returns

True if credit card, false otherwise.

`string_utils.validation.`**`is_decimal`**(*input_string: str*) → bool

Checks whether the given string represents a decimal or not.

A decimal may be signed or unsigned or use a "scientific notation".

```
>>> is_decimal('42.0') # returns true
>>> is_decimal('42') # returns false
```

Parameters

input_string (*str*) – String to check

Returns

True if integer, false otherwise

`string_utils.validation.`**`is_email`**(*input_string: Any*) → bool

Check if a string is a valid email.

Reference: https://tools.ietf.org/html/rfc3696#section-3

Examples:

```
>>> is_email('my.email@the-provider.com') # returns true
>>> is_email('@gmail.com') # returns false
```

Parameters

input_string (*str*) – String to check.

Returns

True if email, false otherwise.

`string_utils.validation.`**`is_full_string`**(*input_string: Any*) → bool

Check if a string is not empty (it must contains at least one non space character).

Examples:

```
>>> is_full_string(None) # returns false
>>> is_full_string('') # returns false
>>> is_full_string(' ') # returns false
>>> is_full_string('hello') # returns true
```

Parameters
input_string (*str*) – String to check.

Returns
True if not empty, false otherwise.

`string_utils.validation.`**`is_integer`**(*input_string: str*) → bool

Checks whether the given string represents an integer or not.

An integer may be signed or unsigned or use a "scientific notation".

Examples:

```
>>> is_integer('42') # returns true
>>> is_integer('42.0') # returns false
```

Parameters
input_string (*str*) – String to check

Returns
True if integer, false otherwise

`string_utils.validation.`**`is_ip`**(*input_string: Any*) → bool

Checks if a string is a valid ip (either v4 or v6).

Examples:

```
>>> is_ip('255.200.100.75') # returns true
>>> is_ip('2001:db8:85a3:0000:0000:8a2e:370:7334') # returns true
>>> is_ip('1.2.3') # returns false
```

Parameters
input_string (*str*) – String to check.

Returns
True if an ip, false otherwise.

`string_utils.validation.`**`is_ip_v4`**(*input_string: Any*) → bool

Checks if a string is a valid ip v4.

Examples:

```
>>> is_ip_v4('255.200.100.75') # returns true
>>> is_ip_v4('nope') # returns false (not an ip)
>>> is_ip_v4('255.200.100.999') # returns false (999 is out of range)
```

Parameters
input_string (*str*) – String to check.

Returns
True if an ip v4, false otherwise.

`string_utils.validation.`**`is_ip_v6`**(*input_string: Any*) → bool

Checks if a string is a valid ip v6.

Examples:

```
>>> is_ip_v6('2001:db8:85a3:0000:0000:8a2e:370:7334') # returns true
>>> is_ip_v6('2001:db8:85a3:0000:0000:8a2e:370:?') # returns false (invalid "?")
```

Parameters
input_string (*str*) – String to check.

Returns
True if a v6 ip, false otherwise.

`string_utils.validation.`**`is_isbn`**(*input_string: str*, *normalize: bool = True*) → bool

Checks if the given string represents a valid ISBN (International Standard Book Number). By default hyphens in the string are ignored, so digits can be separated in different ways, by calling this function with *normalize=False* only digit-only strings will pass the validation.

Examples:

```
>>> is_isbn('9780312498580') # returns true
>>> is_isbn('1506715214') # returns true
```

Parameters

- **input_string** – String to check.
- **normalize** – True to ignore hyphens ("-") in the string (default), false otherwise.

Returns
True if valid ISBN (10 or 13), false otherwise.

`string_utils.validation.`**`is_isbn_10`**(*input_string: str*, *normalize: bool = True*) → bool

Checks if the given string represents a valid ISBN 10 (International Standard Book Number). By default hyphens in the string are ignored, so digits can be separated in different ways, by calling this function with *normalize=False* only digit-only strings will pass the validation.

Examples:

```
>>> is_isbn_10('1506715214') # returns true
>>> is_isbn_10('150-6715214') # returns true
>>> is_isbn_10('150-6715214', normalize=False) # returns false
```

Parameters

- **input_string** – String to check.
- **normalize** – True to ignore hyphens ("-") in the string (default), false otherwise.

Returns
True if valid ISBN 10, false otherwise.

`string_utils.validation.`**`is_isbn_13`**(*input_string: str*, *normalize: bool = True*) → bool

Checks if the given string represents a valid ISBN 13 (International Standard Book Number). By default hyphens in the string are ignored, so digits can be separated in different ways, by calling this function with *normalize=False* only digit-only strings will pass the validation.

Examples:

```
>>> is_isbn_13('9780312498580') # returns true
>>> is_isbn_13('978-0312498580') # returns true
>>> is_isbn_13('978-0312498580', normalize=False) # returns false
```

Parameters

- **input_string** – String to check.
- **normalize** – True to ignore hyphens ("-") in the string (default), false otherwise.

Returns

True if valid ISBN 13, false otherwise.

`string_utils.validation.`**`is_isogram`**(*input_string: Any*) → bool

Checks if the string is an isogram (https://en.wikipedia.org/wiki/Isogram).

Examples:

```
>>> is_isogram('dermatoglyphics') # returns true
>>> is_isogram('hello') # returns false
```

Parameters

input_string (*str*) – String to check.

Returns

True if isogram, false otherwise.

`string_utils.validation.`**`is_json`**(*input_string: Any*) → bool

Check if a string is a valid json.

Examples:

```
>>> is_json('{"name": "Peter"}') # returns true
>>> is_json('[1, 2, 3]') # returns true
>>> is_json('{nope}') # returns false
```

Parameters

input_string (*str*) – String to check.

Returns

True if json, false otherwise

`string_utils.validation.`**`is_number`**(*input_string: str*) → bool

Checks if a string is a valid number.

The number can be a signed (eg: +1, -2, -3.3) or unsigned (eg: 1, 2, 3.3) integer or double or use the "scientific notation" (eg: 1e5).

Examples:

```
>>> is_number('42') # returns true
>>> is_number('19.99') # returns true
>>> is_number('-9.12') # returns true
>>> is_number('1e3') # returns true
>>> is_number('1 2 3') # returns false
```

Parameters
input_string (*str*) – String to check

Returns
True if the string represents a number, false otherwise

`string_utils.validation.`**`is_palindrome`**(*input_string: Any*, *ignore_spaces: bool = False*, *ignore_case: bool = False*) → bool

Checks if the string is a palindrome (https://en.wikipedia.org/wiki/Palindrome).

Examples:

```
>>> is_palindrome('LOL') # returns true
>>> is_palindrome('Lol') # returns false
>>> is_palindrome('Lol', ignore_case=True) # returns true
>>> is_palindrome('ROTFL') # returns false
```

Parameters

- **input_string** (*str*) – String to check.
- **ignore_spaces** (*bool*) – False if white spaces matter (default), true otherwise.
- **ignore_case** (*bool*) – False if char case matters (default), true otherwise.

Returns
True if the string is a palindrome (like "otto", or "i topi non avevano nipoti" if strict=False), False otherwise

`string_utils.validation.`**`is_pangram`**(*input_string: Any*) → bool

Checks if the string is a pangram (https://en.wikipedia.org/wiki/Pangram).

Examples:

```
>>> is_pangram('The quick brown fox jumps over the lazy dog') # returns true
>>> is_pangram('hello world') # returns false
```

Parameters
input_string (*str*) – String to check.

Returns
True if the string is a pangram, False otherwise.

`string_utils.validation.`**`is_slug`**(*input_string: Any*, *separator: str = '-'*) → bool

Checks if a given string is a slug (as created by *slugify()*).

Examples:

```
>>> is_slug('my-blog-post-title') # returns true
>>> is_slug('My blog post title') # returns false
```

Parameters

- **input_string** (*str*) – String to check.
- **separator** (*str*) – Join sign used by the slug.

Returns
True if slug, false otherwise.

`string_utils.validation.`**`is_snake_case`**(*input_string: Any*, *separator: str = '_'*) → bool

Checks if a string is formatted as "snake case".

A string is considered snake case when:

- it's composed only by lowercase/uppercase letters and digits
- it contains at least one underscore (or provided separator)
- it does not start with a number

Examples:

```
>>> is_snake_case('foo_bar_baz') # returns true
>>> is_snake_case('foo') # returns false
```

Parameters

- **input_string** (*str*) – String to test.
- **separator** (*str*) – String to use as separator.

Returns
True for a snake case string, false otherwise.

`string_utils.validation.`**`is_string`**(*obj: Any*) → bool

Checks if an object is a string.

Example:

```
>>> is_string('foo') # returns true
>>> is_string(b'foo') # returns false
```

Parameters
obj – Object to test.

Returns
True if string, false otherwise.

`string_utils.validation.`**`is_url`**(*input_string: Any*, *allowed_schemes: List[str] | None = None*) → bool

Check if a string is a valid url.

Examples:

```
>>> is_url('http://www.mysite.com') # returns true
>>> is_url('https://mysite.com') # returns true
>>> is_url('.mysite.com') # returns false
```

Parameters

- **input_string** (*str*) – String to check.

- **allowed_schemes** (*Optional[List[str]]*) – List of valid schemes ('http', 'https', 'ftp'...). Default to None (any scheme is valid).

Returns
True if url, false otherwise

`string_utils.validation.`**`is_uuid`**(*input_string: Any*, *allow_hex: bool = False*) → bool

Check if a string is a valid UUID.

Example:

```
>>> is_uuid('6f8aa2f9-686c-4ac3-8766-5712354a04cf') # returns true
>>> is_uuid('6f8aa2f9686c4ac387665712354a04cf') # returns false
>>> is_uuid('6f8aa2f9686c4ac387665712354a04cf', allow_hex=True) # returns true
```

Parameters

- **input_string** (*str*) – String to check.
- **allow_hex** (*bool*) – True to allow UUID hex representation as valid, false otherwise (default)

Returns
True if UUID, false otherwise

`string_utils.validation.`**`words_count`**(*input_string: str*) → int

Returns the number of words contained into the given string.

This method is smart, it does consider only sequence of one or more letter and/or numbers as "words", so a string like this: "! @ # % ... []" will return zero! Moreover it is aware of punctuation, so the count for a string like "one,two,three.stop" will be 4 not 1 (even if there are no spaces in the string).

Examples:

```
>>> words_count('hello world') # returns 2
>>> words_count('one,two,three.stop') # returns 4
```

Parameters
input_string (*str*) – String to check.

Returns
Number of words.

4.2 String Manipulation

`string_utils.manipulation.`**`asciify`**(*input_string: str*) → str

Force string content to be ascii-only by translating all non-ascii chars into the closest possible representation (eg: ó -> o, Ë -> E, ç -> c...).

Bear in mind: Some chars may be lost if impossible to translate.

Example:

```
>>> asciify('èéùúòóäåëýñÅÀÁÇÌÍÑÓË') # returns 'eeuuooaaeynAAACIINOE'
```

Parameters
input_string – String to convert

Returns
Ascii utf-8 string

`string_utils.manipulation.booleanize(input_string: str) → bool`

Turns a string into a boolean based on its content (CASE INSENSITIVE).

A positive boolean (True) is returned if the string value is one of the following:

- "true"
- "1"
- "yes"
- "y"

Otherwise False is returned.

Examples:

```
>>> booleanize('true') # returns True
>>> booleanize('YES') # returns True
>>> booleanize('nope') # returns False
```

Parameters
input_string (*str*) – String to convert

Returns
True if the string contains a boolean-like positive value, false otherwise

`string_utils.manipulation.camel_case_to_snake(input_string, separator='_')`

Convert a camel case string into a snake case one. (The original string is returned if is not a valid camel case string)

Example:

```
>>> camel_case_to_snake('ThisIsACamelStringTest') # returns 'this_is_a_camel_case_
↪string_test'
```

Parameters

- **input_string** (*str*) – String to convert.
- **separator** (*str*) – Sign to use as separator.

Returns
Converted string.

`string_utils.manipulation.compress(input_string: str, encoding: str = 'utf-8', compression_level: int = 9) → str`

Compress the given string by returning a shorter one that can be safely used in any context (like URL) and restored back to its original state using *decompress()*.

Bear in mind: Besides the provided *compression_level*, the compression result (how much the string is actually compressed by resulting into a shorter string) depends on 2 factors:

1. The amount of data (string size): short strings might not provide a significant compression result or even be longer than the given input string (this is due to the fact that some bytes have to be embedded into the compressed string in order to be able to restore it later on)

2. The content type: random sequences of chars are very unlikely to be successfully compressed, while the best compression result is obtained when the string contains several recurring char sequences (like in the example).

Behind the scenes this method makes use of the standard Python's zlib and base64 libraries.

Examples:

```
>>> n = 0 # <- ignore this, it's a fix for Pycharm (not fixable using ignore␣
↪comments)
>>> # "original" will be a string with 169 chars:
>>> original = ' '.join(['word n{}'.format(n) for n in range(20)])
>>> # "compressed" will be a string of 88 chars
>>> compressed = compress(original)
```

Parameters

- **input_string** (*str*) – String to compress (must be not empty or a ValueError will be raised).
- **encoding** (*str*) – String encoding (default to "utf-8").
- **compression_level** (*int*) – A value between 0 (no compression) and 9 (best compression), default to 9.

Returns
Compressed string.

`string_utils.manipulation.`**`decompress`**(*input_string: str*, *encoding: str = 'utf-8'*) → str

Restore a previously compressed string (obtained using *compress()*) back to its original state.

Parameters

- **input_string** (*str*) – String to restore.
- **encoding** (*str*) – Original string encoding.

Returns
Decompressed string.

`string_utils.manipulation.`**`prettify`**(*input_string: str*) → str

Reformat a string by applying the following basic grammar and formatting rules:

- String cannot start or end with spaces
- The first letter in the string and the ones after a dot, an exclamation or a question mark must be uppercase
- String cannot have multiple sequential spaces, empty lines or punctuation (except for "?", "!" and ".")
- Arithmetic operators (+, -, /, *, =) must have one, and only one space before and after themselves
- One, and only one space should follow a dot, a comma, an exclamation or a question mark
- Text inside double quotes cannot start or end with spaces, but one, and only one space must come first and after quotes (foo" bar"baz -> foo "bar" baz)
- Text inside round brackets cannot start or end with spaces, but one, and only one space must come first and after brackets ("foo(bar)baz" -> "foo (bar) baz")
- Percentage sign ("%") cannot be preceded by a space if there is a number before ("100 %" -> "100%")
- Saxon genitive is correct ("Dave' s dog" -> "Dave's dog")

Examples:

```
>>> prettify(' unprettified string ,, like this one,will be"prettified" .it\' s␣
↪awesome! ')
>>> # -> 'Unprettified string, like this one, will be "prettified". It's awesome!'
```

Parameters
input_string – String to manipulate

Returns
Prettified string.

`string_utils.manipulation.`**`reverse`**`(`*input_string: str*`)` → str

Returns the string with its chars reversed.

Example:

```
>>> reverse('hello') # returns 'olleh'
```

Parameters
input_string (*str*) – String to revert.

Returns
Reversed string.

`string_utils.manipulation.`**`roman_decode`**`(`*input_string: str*`)` → int

Decode a roman number string into an integer if the provided string is valid.

Example:

```
>>> roman_decode('VII') # returns 7
```

Parameters
input_string (*str*) – (Assumed) Roman number

Returns
Integer value

`string_utils.manipulation.`**`roman_encode`**`(`*input_number: str | int*`)` → str

Convert the given number/string into a roman number.

The passed input must represents a positive integer in the range 1-3999 (inclusive).

Why this limit? You may be wondering:

1. zero is forbidden since there is no related representation in roman numbers
2. the upper bound 3999 is due to the limitation in the ascii charset (the higher quantity sign displayable in ascii is "M" which is equal to 1000, therefore based on roman numbers rules we can use 3 times M to reach 3000 but we can't go any further in thousands without special "boxed chars").

Examples:

```
>>> roman_encode(37) # returns 'XXXVIII'
>>> roman_encode('2020') # returns 'MMXX'
```

Parameters
input_number (*Union[str, int]*) – An integer or a string to be converted.

Returns
Roman number string.

`string_utils.manipulation.shuffle(input_string: str) → str`

Return a new string containing same chars of the given one but in a randomized order.

Example:

```
>>> shuffle('hello world') # possible output: 'l wodheorll'
```

Parameters
input_string (*str*) – String to shuffle

Returns
Shuffled string

`string_utils.manipulation.slugify(input_string: str, separator: str = '-') → str`

Converts a string into a "slug" using provided separator. The returned string has the following properties:

- it has no spaces
- all letters are in lower case
- all punctuation signs and non alphanumeric chars are removed
- words are divided using provided separator
- all chars are encoded as ascii (by using *asciify()*)
- is safe for URL

Examples:

```
>>> slugify('Top 10 Reasons To Love Dogs!!!') # returns: 'top-10-reasons-to-love-dogs
↪'
>>> slugify('Mönstér Mägnët') # returns 'monster-magnet'
```

Parameters

- **input_string** (*str*) – String to convert.
- **separator** (*str*) – Sign used to join string tokens (default to "-").

Returns
Slug string

`string_utils.manipulation.snake_case_to_camel(input_string: str, upper_case_first: bool = True, separator: str = '_') → str`

Convert a snake case string into a camel case one. (The original string is returned if is not a valid snake case string)

Example:

```
>>> snake_case_to_camel('the_snake_is_green') # returns 'TheSnakeIsGreen'
```

Parameters

- **input_string** (*str*) – String to convert.
- **upper_case_first** (*bool*) – True to turn the first letter into uppercase (default).

- **separator** (*str*) – Sign to use as separator (default to "_").

Returns
Converted string

`string_utils.manipulation.strip_html(input_string: str, keep_tag_content: bool = False)` → str

Remove html code contained into the given string.

Examples:

```
>>> strip_html('test: <a href="foo/bar">click here</a>') # returns 'test: '
>>> strip_html('test: <a href="foo/bar">click here</a>', keep_tag_content=True) #␣
↪returns 'test: click here'
```

Parameters

- **input_string** (*str*) – String to manipulate.
- **keep_tag_content** (*bool*) – True to preserve tag content, False to remove tag and its content too (default).

Returns
String with html removed.

`string_utils.manipulation.strip_margin(input_string: str)` → str

Removes tab indentation from multi line strings (inspired by analogous Scala function).

Example:

```
>>> strip_margin('''
>>>                 line 1
>>>                 line 2
>>>                 line 3
>>> ''')
>>> # returns:
>>> '''
>>> line 1
>>> line 2
>>> line 3
>>> '''
```

Parameters
input_string (*str*) – String to format

Returns
A string without left margins

4.3 String Generation

`string_utils.generation.random_string(size: int)` → str

Returns a string of the specified size containing random characters (uppercase/lowercase ascii letters and digits).

Example:

```
>>> random_string(9) # possible output: "cx3QQbzYg"
```

Parameters
size (*int*) – Desired string size

Returns
Random string

`string_utils.generation.roman_range(stop: int, start: int = 1, step: int = 1)` → Generator

Similarly to native Python's *range()*, returns a Generator object which generates a new roman number on each iteration instead of an integer.

Example:

```
>>> for n in roman_range(7): print(n)
>>> # prints: I, II, III, IV, V, VI, VII
>>> for n in roman_range(start=7, stop=1, step=-1): print(n)
>>> # prints: VII, VI, V, IV, III, II, I
```

Parameters

- **stop** – Number at which the generation must stop (must be <= 3999).
- **start** – Number at which the generation must start (must be >= 1).
- **step** – Increment of each generation step (default to 1).

Returns
Generator of roman numbers.

`string_utils.generation.secure_random_hex(byte_count: int)` → str

Generates a random string using secure low level random generator (os.urandom).

Bear in mind: due to hex conversion, the returned string will have a size that is exactly the double of the given *byte_count*.

Example:

```
>>> secure_random_hex(9) # possible output: 'aac4cf1d1d87bd5036'
```

Parameters
byte_count (*int*) – Number of random bytes to generate

Returns
Hexadecimal string representation of generated random bytes

`string_utils.generation.uuid(as_hex: bool = False)` → str

Generated an UUID string (using *uuid.uuid4()*).

Examples:

```
>>> uuid() # possible output: '97e3a716-6b33-4ab9-9bb1-8128cb24d76b'
>>> uuid(as_hex=True) # possible output: '97e3a7166b334ab99bb18128cb24d76b'
```

Parameters
as_hex – True to return the hex value of the UUID, False to get its default representation (default).

Returns
uuid string.

4.4 Errors

exception `string_utils.errors.`**`InvalidInputError`**(*input_data: Any*)

Bases: `TypeError`

Custom error raised when received object is not a string as expected.

CHAPTER

FIVE

FUNCTION INDICES

- genindex

PYTHON MODULE INDEX

S

string_utils.errors, 23
string_utils.generation, 21
string_utils.manipulation, 16
string_utils.validation, 9

INDEX

A

asciify() (*in module string_utils.manipulation*), 16

B

booleanize() (*in module string_utils.manipulation*), 17

C

camel_case_to_snake() (*in module string_utils.manipulation*), 17
compress() (*in module string_utils.manipulation*), 17
contains_html() (*in module string_utils.validation*), 9

D

decompress() (*in module string_utils.manipulation*), 18

I

InvalidInputError, 23
is_camel_case() (*in module string_utils.validation*), 9
is_credit_card() (*in module string_utils.validation*), 9
is_decimal() (*in module string_utils.validation*), 10
is_email() (*in module string_utils.validation*), 10
is_full_string() (*in module string_utils.validation*), 10
is_integer() (*in module string_utils.validation*), 11
is_ip() (*in module string_utils.validation*), 11
is_ip_v4() (*in module string_utils.validation*), 11
is_ip_v6() (*in module string_utils.validation*), 12
is_isbn() (*in module string_utils.validation*), 12
is_isbn_10() (*in module string_utils.validation*), 12
is_isbn_13() (*in module string_utils.validation*), 13
is_isogram() (*in module string_utils.validation*), 13
is_json() (*in module string_utils.validation*), 13
is_number() (*in module string_utils.validation*), 13
is_palindrome() (*in module string_utils.validation*), 14
is_pangram() (*in module string_utils.validation*), 14
is_slug() (*in module string_utils.validation*), 14
is_snake_case() (*in module string_utils.validation*), 15
is_string() (*in module string_utils.validation*), 15
is_url() (*in module string_utils.validation*), 15
is_uuid() (*in module string_utils.validation*), 16

M

module
    string_utils.errors, 23
    string_utils.generation, 21
    string_utils.manipulation, 16
    string_utils.validation, 9

P

prettify() (*in module string_utils.manipulation*), 18

R

random_string() (*in module string_utils.generation*), 21
reverse() (*in module string_utils.manipulation*), 19
roman_decode() (*in module string_utils.manipulation*), 19
roman_encode() (*in module string_utils.manipulation*), 19
roman_range() (*in module string_utils.generation*), 22

S

secure_random_hex() (*in module string_utils.generation*), 22
shuffle() (*in module string_utils.manipulation*), 20
slugify() (*in module string_utils.manipulation*), 20
snake_case_to_camel() (*in module string_utils.manipulation*), 20
string_utils.errors
    module, 23
string_utils.generation
    module, 21
string_utils.manipulation
    module, 16
string_utils.validation
    module, 9
strip_html() (*in module string_utils.manipulation*), 21
strip_margin() (*in module string_utils.manipulation*), 21

U

`uuid()` (*in module string_utils.generation*), 22

W

`words_count()` (*in module string_utils.validation*), 16