blur Documentation

Release 0.5-dev

Andrew Yoon

Oct 24, 2016

Contents

a simple python toolkit for chance operations in art.

Contents:

CHAPTER 1

Introduction

blur is a suite of tools for Python to help make using chance operations in algorithmic art easier.

```
>>> from blur.markov.graph import Graph
>>> word_distance_weights = {-5: 1, -1: 2, 1: 8, 3: 3}
>>> graph = Graph.from_string('blur is a suite of tools to help make using '
... 'chance operations in algorithmic art easier', word_distance_weights)
>>> print(' '.join(graph.pick().value for i in range(10)))
using chance algorithmic in algorithmic art easier blur easier blur
```

blur allows you to easily implement complex chance generated elements in your works without getting bogged down in the boilerplate and the nitty-gritty math.

```
>>> from blur import soft, rand
>>> frequency_profile = rand.normal_distribution(mean=440, variance=50)
>>> for i in range(100):
...     add_oscillator(frequency=weighted_rand(frequency_profile))
# ------------------------------------------------------------------------
>>> color = soft.SoftColor(red=([(255, 10), (100, 0)],),
...                        green=([(200, 10), (100, 0)],),
...                        blue=([(0, 20), (80, 1)],))
>>> apple.fill_color_rgb = color.get()
# ------------------------------------------------------------------------
>>> words = soft.SoftOptions([('nothing', 10),
...                           ('something', 3),
...                           ('everything', 1)])
>>> print('I have {0} to say and I am saying it and that is poetry.'.format(
...     words.get())
```

At the heart of *blur* is the `rand` module, which contains a series of functions for finding non-uniform random numbers based on customizable weights. *blur* also comes with a model markov graph, a tool for deriving markov graphs from text, a collection of soft objects whose values vary according to customizable weights and rules, and a model I Ching.

Besides Python (version 2.7 or 3.3), *blur* has no dependencies, making it easy to incorporate into your project.

To install *blur* use pip from the command line:

```
$ pip install blur
```

CHAPTER 2

Installation

Stable

blur has no external dependencies and the latest stable version can be easily installed with `pip` from the command line

```
$ pip install blur
```

Latest

To install the latest development version clone the upstream repository, ensure you are on the `dev` branch, and run the `setup.py` script

```
$ git clone git@github.com:ajyoon/blur.git
$ cd blur
blur$ git status
On branch dev
Your branch is up-to-date with 'origin/dev'.
nothing to commit, working tree clean
blur$ python setup.py install
```

CHAPTER 3

API documentation

Contents:

3.1 rand

A collection of functions for performing non-uniform random operations.

Many functions rely of weight 2-tuples of the form `(outcome,strength)` where `strength` is a number and `outcome` is either a number or any type depending on the function.

exception `blur.rand.`**`ProbabilityUndefinedError`**
Exception raised when the probability of a system is undefined.

`blur.rand.`**`bound_weights`**(*weights*, *minimum=None*, *maximum=None*)
Bound a weight list so that all outcomes fit within specified bounds.

The probability distribution within the `minimum` and `maximum` values remains the same. Weights in the list with outcomes outside of `minimum` and `maximum` are removed. If weights are removed from either end, attach weights at the modified edges at the same weight (y-axis) position they had interpolated in the original list.

If neither `minimum` nor `maximum` are set, `weights` will be returned unmodified. If both are set, `minimum` must be less than `maximum`.

Parameters

- **`weights`** (*list*) – the list of weights where each weight is a `tuple` of form `(float,float)` corresponding to `(outcome,weight)`. Must be sorted in increasing order of outcomes
- **`minimum`** (*float*) – Lowest allowed outcome for the weight list
- **`maximum`** (*float*) – Highest allowed outcome for the weight list

Returns *list* – A list of 2-tuples of form `(float,float)`, the bounded weight list.

Raises `ValueError` – if `maximum < minimum`

Example

```
>>> weights = [(0, 0), (2, 2), (4, 0)]
>>> bound_weights(weights, 1, 3)
[(1, 1), (2, 2), (3, 1)]
```

blur.rand.**normal_distribution**(*mean*, *variance*, *minimum=None*, *maximum=None*, *weight_count=23*)

Return a list of weights approximating a normal distribution.

Parameters

- **mean** (*float*) – The mean of the distribution
- **variance** (*float*) – The variance of the distribution
- **minimum** (*float*) – The minimum outcome possible to bound the output distribution to
- **maximum** (*float*) – The maximum outcome possible to bound the output distribution to
- **weight_count** (*int*) – The number of weights that will be used to approximate the distribution

Returns *list* – a list of `(float,float)` weight tuples approximating a normal distribution.

Raises

- `ValueError` – `if maximum < minimum`
- `TypeError` – if both `minimum` and `maximum` are `None`

Example

```
>>> weights = normal_distribution(10, 3,
...                               minimum=0, maximum=20,
...                               weight_count=5)
>>> rounded_weights = [(round(value, 2), round(strength, 2))
...                    for value, strength in weights]
>>> rounded_weights
[(1.34, 0.0), (4.8, 0.0), (8.27, 0.14), (11.73, 0.14), (15.2, 0.0)]
```

blur.rand.**prob_bool**(*probability*)

Return `True` or `False` depending on `probability`.

Parameters **probability** (*float*) – Probability between `0` and `1` to return `True` where `0` is guaranteed to return `False` and `1` is guaranteed to return `True`.

Returns *bool* – `True` or `False` depending on `probability`.

Example

```
>>> prob_bool(0.9)
# Usually will be...
True
>>> prob_bool(0.1)
# Usually will be...
False
```

blur.rand.**percent_possible**(*percent*)

Return True `percent` / 100 times.

Parameters **percent** (*int or float*) – percent possibility to return True

Returns *bool* – Either `True` or `False` depending on `percent`

Example

```
>>> percent_possible(90)
# Usually will be...
True
>>> percent_possible(10)
# Usually will be...
False
```

blur.rand.**pos_or_neg**(*value*, *prob_pos=0.5*)

Return either positive or negative `value` based on `prob_pos`.

This is equivalent to `abs(value) * pos_or_neg_1(prob_pos)`.

Parameters

- **value** (*int or float*) – the value to operate on
- **prob_pos** (*float*) – The probability to return positive. where `prob_pos = 0` is guaranteed to return negative and `prob_pos = 1` is guaranteed to return positive. Default value is `0.5`.

Returns *int or float* – `value` either positive or negative

Example

```
>>> pos_or_neg(42, 0.9)
# Usually will be...
42
>>> pos_or_neg(42, 0.1)
# Usually will be...
-42
```

blur.rand.**pos_or_neg_1**(*prob_pos=0.5*)

Return either `1` with probability of `prob_pos`, otherwise `-1`.

Parameters **prob_pos** (*float*) – The probability to return positive `1` where `prob_pos = 0` is guaranteed to return negative and `prob_pos = 1` is guaranteed to return positive. Default value is `0.5`.

Returns *int* – either `1` or `-1`

Example

```
>>> pos_or_neg_1(0.9)
# Usually will be...
1
>>> pos_or_neg_1(0.1)
# Usually will be...
-1
```

blur.rand.**weighted_rand**(*weights*, *round_result=False*)

Generate a non-uniform random value based on a list of weight tuples.

Treats weights as coordinates for a probability distribution curve and rolls accordingly. Constructs a piece-wise linear curve according to coordinates given in `weights` and rolls random values in the curve's bounding box until a value is found under the curve

Weight tuples should be of the form: (outcome, strength).

Parameters

- **weights** – (list): the list of weights where each weight is a tuple of form `(float,float)` corresponding to `(outcome,strength)`. Weights with strength `0` or less will have no chance to be rolled. The list must be sorted in increasing order of outcomes.
- **round_result** (*bool*) – Whether or not to round the resulting value to the nearest integer.

Returns

float – A weighted random number

int: A weighted random number rounded to the nearest `int`

Example

```
>>> weighted_rand([(-3, 4), (0, 10), (5, 1)])
-0.650612268193731
>>> weighted_rand([(-3, 4), (0, 10), (5, 1)])
-2
```

blur.rand.**weighted_choice**(*weights*, *as_index_and_value_tuple=False*)

Generate a non-uniform random choice based on a list of option tuples.

Treats each outcome as a discreet unit with a chance to occur.

Parameters

- **weights** (*list*) – a list of options where each option is a tuple of form `(Any,float)` corresponding to `(outcome,strength)`. Outcome values may be of any type. Options with strength `0` or less will have no chance to be chosen.
- **as_index_and_value_tuple** (*bool*) – Option to return an `(index,value)` tuple instead of just a single `value`. This is useful when multiple outcomes in `weights` are the same and you need to know exactly which one was picked.

Returns

Any – If `as_index_and_value_tuple is False`, any one of the items in the outcomes of `weights`

tuple (int, Any): If `as_index_and_value_tuple is True`, a 2-tuple of form `(int,Any)` corresponding to `(index,value)`. the index as well as value of the item that was picked.

Example

```
>>> choices = [('choice one', 10), ('choice two', 3)]
>>> weighted_choice(choices)
# Often will be...
'choice one'
```

```
>>> weighted_choice(choices,
...                 as_index_and_value_tuple=True)
# Often will be...
(0, 'choice one')
```

blur.rand.**weighted_order**(*weights*)

Non-uniformally order a list according to weighted priorities.

`weights` is a list of tuples of form `(Any,float or int)` corresponding to `(item,strength)`. The output list is constructed by repeatedly calling `weighted_choice()` on the weights, adding items to the end of the list as they are picked.

Higher strength weights will have a higher chance of appearing near the beginning of the output list.

A list weights with uniform strengths is equivalent to calling `random.shuffle()` on the list of items.

If any weight strengths are `<= 0`, a `ProbabilityUndefinedError` is be raised.

Passing an empty list will return an empty list.

Parameters **weights** (*list*) – a list of tuples of form `(Any,float or int)` corresponding to `(item,strength)`. The output list is constructed by repeatedly calling `weighted_choice()` on the weights, appending items to the output list as they are picked.

Returns *list* – the newly ordered list

Raises *ProbabilityUndefinedError* – if any weight's strength is below 0.

Example

```
>>> weights = [('Probably Earlier', 100),
...            ('Probably Middle', 20),
...            ('Probably Last', 1)]
>>> weighted_order(weights)
['Probably Earlier', 'Probably Middle', 'Probably Last']
```

3.2 soft

A collection of soft objects.

Every soft object has a value that changes every time it is retrieved according to defined chance profiles. This value can be retrieved with the `SoftObject` 's `get()` method.

```
>>> blurry_float = SoftFloat([(-1, 2), (3, 5)])
>>> blurry_float.get()
1.925674784815838
>>> blurry_float.get()
1.120389067727415
>>> blurry_float.get()
1.30418962132812
```

class blur.soft.**SoftObject**

An abstract base class for `SoftObject` 's.

Direct instances of `SoftObject` should not be created; instead, the appropriate subclass should be used.

Every SoftObject represents a stochastic blurry object whose value is determined with the `get()` method.

This is an abstract method and should not be called. Subclasses of `SoftObject` must override and implement this.

Direct instances of `SoftObject` should not be created; instead, the appropriate subclass should be used.

get()
Retrieve a value of this `SoftObject`.

This is an abstract method and should not be called. Subclasses of `SoftObject` must override and implement this.

class `blur.soft.`**SoftOptions**(*options*)
One of many objects with corresponding weights.

Parameters **options** (*list*) – a list of options where each option is a `tuple` of form `(Any,float)` corresponding to `(outcome,weight)`. Outcome values may be of any type. Weights `0` or less will have no chance to be retrieved by `get()`

Example

```
>>> options = SoftOptions([('option one', 2),
...                        ('option two', 5),
...                        ('option three', 8)])
>>> options.get()
'option three'
```

classmethod **with_uniform_weights**(*options*, *weight=1*)
Initialize from a list of options, assigning uniform weights.

Parameters

- **options** (*list*) – The list of options of any type this object can return with the `get()` method.
- **weight** (*float or int*) – The weight to be assigned to every option. Regardless of what this is, the probability of each option will be the same. In almost all cases this can be ignored. The only case for explicitly setting this is if you need to modify the weights after creation with specific requirements.

Returns *SoftOptions* – A newly constructed instance

Example

```
>>> blurry_object = SoftOptions.with_uniform_weights(
...     ['option one', 'option two', 'option three'])
>>> blurry_object.options
[('option one', 1), ('option two', 1), ('option three', 1)]
```

classmethod **with_random_weights**(*options*)
Initialize from a list of options with random weights.

The weights assigned to each object are uniformally random integers between `1` and `len(options)`

Parameters **options** (*list*) – The list of options of any type this object can return with the `get()` method.

Returns *SoftOptions* – A newly constructed instance

options
list – a list of options where each option is a `tuple` of form `(Any,float or int)` corresponding to `(outcome,weight)`. Outcome values may be of any type. Weights `0` or less will have no chance to be retrieved by `get()`

get()
Get one of the options within the probability space of the object.

Returns *Any* – An item from `self.options`.

class `blur.soft.`**SoftBool**(*prob_true*)
A stochastic `bool` defined by a probability to be `True`.

Parameters **prob_true** (*float*) – The probability that `get()` returns `True` where `prob_true <= 0` is always `False` and `prob_true >= 1` is always `True`.

prob_true
float or int – The probability that `get()` returns `True` where `prob_true <= 0` is always `False` and `prob_true >= 1` is always `True`.

get()
Get either `True` or `False` depending on `self.prob_true`.

Returns *bool* – `True` or `False` depending on `self.prob_true`.

class `blur.soft.`**SoftFloat**(*weights*)
A stochastic float value defined by a list of weights.

Parameters **weights** (*list*) – the list of weights where each weight is a tuple of form `(int or float,int or float)` corresponding to `(outcome,strength)`. These weights represent the stochastic value of this *SoftFloat*.

classmethod **bounded_uniform**(*lowest*, *highest*, *weight_interval=None*)
Initialize with a uniform distribution between two values.

If no `weight_interval` is passed, this weight distribution will just consist of `[(lowest,1),(highest,1)]`. If specified, weights (still with uniform weight distribution) will be added every `weight_interval`. Use this if you intend to modify the weights in any complex way after initialization.

Parameters

- **lowest** (*float or int*) –
- **highest** (*float or int*) –
- **weight_interval** (*int*) –

Returns *SoftFloat* – A newly constructed instance.

weights
list – the list of weights where each weight is a tuple of form `(int or float,int or float)` corresponding to `(outcome,strength)`. These weights represent the stochastic value of this *SoftFloat*.

get()
Get a `float` value in the probability space of the object.

Returns *float* – A value between the lowest and highest outcomes in `self.weights`

class `blur.soft.`**SoftInt**(*weights*)
A stochastic `int` value defined by a list of weights.

Has the exact same functionality as `SoftFloat`, except that `get()` returns `int` values

Parameters **weights** (*list*) – the list of weights where each weight is a tuple of form `(int or float,int or float)` corresponding to `(outcome,strength)`. These weights represent the stochastic value of this *SoftFloat*.

get()
Get an `int` value in the probability space of the object.

Returns: int

bounded_uniform(*lowest*, *highest*, *weight_interval=None*)
Initialize with a uniform distribution between two values.

If no `weight_interval` is passed, this weight distribution will just consist of `[(lowest,1),(highest,1)]`. If specified, weights (still with uniform weight distribution) will be added every `weight_interval`. Use this if you intend to modify the weights in any complex way after initialization.

Parameters

- **lowest** (*float or int*) –
- **highest** (*float or int*) –
- **weight_interval** (*int*) –

Returns *SoftFloat* – A newly constructed instance.

weights
list – the list of weights where each weight is a tuple of form `(int or float,int or float)` corresponding to `(outcome,strength)`. These weights represent the stochastic value of this *SoftFloat*.

class `blur.soft.`**SoftColor**(*red*, *green*, *blue*)
An RGB color whose individual channels can be `SoftInt` objects.

`SoftColor.get()` returns an `(r,g,b)` tuple o integers. To get a hexadecimal color value, use `get_as_hex()`.

```
>>> color = SoftColor(234,                          # static red
...                   124,                          # static green
...                   SoftInt([(0, 10), (40, 20)]))  # soft blue
>>> rgb = color.get()
>>> rgb
(234, 124, 32)
# Conveniently convert the value to hex
>>> SoftColor.rgb_to_hex(rgb)
'#EA7C20'
# Generate a new value directly as hex
>>> some_soft_color.get_as_hex()
'#EA7C20'
```

Parameters

- **red** (*int or SoftInt or tuple(args for SoftInt)*) –
- **green** (*int or SoftInt or tuple(args for SoftInt)*) –
- **blue** (*int or SoftInt or tuple(args for SoftInt)*) –

Raises `TypeError` – if invalid types are passed in args

Notes

When initializing the soft color channels using the convenience option to pass a tuple of args for for `SoftInt.__init__()`, keep in mind that when creating 1-length tuples in Python you need to add a comma after the first element, or Python will ignore the parentheses.

```
color = SoftColor(([(0, 1), (255, 10)],),
                  ([(0, 1), (255, 10)],),
                  ([(0, 1), (255, 10)],))
```

red
SoftInt or int – The red channel of the RGB color

green
SoftInt or int – The green channel of the RGB color

blue
SoftInt or int – The blue channel of the RGB color

classmethod rgb_to_hex(*color*)
Convert an `(r,g,b)` color tuple to a hexadecimal string.

Alphabetical characters in the output will be capitalized.

Parameters **color** (*tuple*) – An rgb color tuple of form: (int, int, int)

Returns: string

Example

```
>>> SoftColor.rgb_to_hex((0, 0, 0))
'#000000'
>>> SoftColor.rgb_to_hex((255, 255, 255))
'#FFFFFF'
```

get()
Get an rgb color tuple according to the probability distribution.

Returns **tuple** (*int, int, int*) – A `(red,green,blue)` tuple.

Example

```
>>> color = SoftColor(([(0, 1), (255, 10)],),
...                   ([(0, 1), (255, 10)],),
...                   ([(0, 1), (255, 10)],))
>>> color.get()
(234, 201, 243)
```

get_as_hex()
Get a hexademical color according to the probability distribution.

Equivalent to `SoftColor.rgb_to_hex(self.get())`

Returns *str* – A hexademical color string

Example

```
>>> color = SoftColor(([(0, 1), (255, 10)],),
...                   ([(0, 1), (255, 10)],),
...                   ([(0, 1), (255, 10)],))
>>> color.get_as_hex()
'#C8EABB'
```

3.3 markov

A subpackage containing utilities for building Markov graphs.

Contents:

3.3.1 graph

A module containing a model Markov graph.

```
>>> from blur.markov.node import Node
>>> node_1 = Node('One')
>>> node_2 = Node('Two')
>>> node_1.add_link(node_1, 5)
>>> node_1.add_link(node_2, 2)
>>> node_2.add_link(node_1, 1)
>>> graph = Graph([node_1, node_2])
>>> [graph.pick().get_value() for i in range(10)]
['One', 'One', 'One', 'One', 'One', 'One', 'Two', 'One', 'One', 'One']
```

class `blur.markov.graph.`**`Graph`**(*node_list=None*)

A Markov graph with a number of handy utilities.

The graph consists of a list of `Node` 's and keeps track of which node was picked last.

Several utilities offer conveniences for managing the network.

Parameters **`node_list`** (*list*) – An optional list of nodes to populate the network with. To populate the network after initialization, use the `Graph.add_nodes()` method.

Warning: Nodes are not copied when placed into the graph: the passed nodes are used in the object. Side effects may occur if node-altering methods are called, such as `Graph.apply_noise()` or `Graph.feather_links()`. Handle with care if using the same `Node` in multiple contexts.

`merge_nodes`(*keep_node*, *kill_node*)

Merge two nodes in the graph.

Takes two nodes and merges them together, merging their links by combining the two link lists and summing the weights of links which point to the same node.

All links in the graph pointing to `kill_node` will be merged into `keep_node`.

Links belonging to `kill_node` which point to targets not in `self.node_list` will not be merged into `keep_node`

Parameters

- **keep_node** (Node) – node to be kept
- **kill_node** (Node) – node to be deleted

Returns: None

Example

```
>>> from blur.markov.node import Node
>>> node_1 = Node('One')
>>> node_2 = Node('Two')
>>> node_3 = Node('Three')
>>> node_1.add_link(node_3, 7)
>>> node_2.add_link(node_1, 1)
>>> node_2.add_link(node_2, 3)
>>> node_3.add_link(node_2, 5)
>>> graph = Graph([node_1, node_2, node_3])
>>> print([node.value for node in graph.node_list])
['One', 'Two', 'Three']
>>> graph.merge_nodes(node_2, node_3)
>>> print([node.value for node in graph.node_list])
['One', 'Two']
>>> for link in graph.node_list[1].link_list:
...     print('{} {}'.format(link.target.value, link.weight))
One 1
Two 8
```

add_nodes (*nodes*)

Add a given node or list of nodes to self.node_list.

Parameters **node** (*Node or list[Node]*) – the node or list of nodes to add to the graph

Returns: None

Examples:

Adding one node:

```
>>> from blur.markov.node import Node
>>> graph = Graph()
>>> node_1 = Node('One')
>>> graph.add_nodes(node_1)
>>> print([node.value for node in graph.node_list])
['One']
```

Adding multiple nodes at a time in a list:

```
>>> from blur.markov.node import Node
>>> graph = Graph()
>>> node_1 = Node('One')
>>> node_2 = Node('Two')
>>> graph.add_nodes([node_1, node_2])
>>> print([node.value for node in graph.node_list])
['One', 'Two']
```

feather_links (*factor=0.01*, *include_self=False*)

Feather the links of connected nodes.

Go through every node in the network and make it inherit the links of the other nodes it is connected to. Because the link weight sum for any given node can be very different within a graph, the weights of inherited links are made proportional to the sum weight of the parent nodes.

Parameters

- **factor** (*float*) – multiplier of neighbor links
- **include_self** (*bool*) – whether nodes can inherit links pointing to themselves

Returns: None

Example

```
>>> from blur.markov.node import Node
>>> node_1 = Node('One')
>>> node_2 = Node('Two')
>>> node_1.add_link(node_2, 1)
>>> node_2.add_link(node_1, 1)
>>> graph = Graph([node_1, node_2])
>>> for link in graph.node_list[0].link_list:
...     print('{} {}'.format(link.target.value, link.weight))
Two 1
>>> graph.feather_links(include_self=True)
>>> for link in graph.node_list[0].link_list:
...     print('{} {}'.format(link.target.value, link.weight))
Two 1
One 0.01
```

apply_noise(*noise_weights=None*, *uniform_amount=0.1*)

Add noise to every link in the network.

Can use either a `uniform_amount` or a `noise_weight` weight profile. If `noise_weight` is set, `uniform_amount` will be ignored.

Parameters

- **noise_weights** (*list*) – a list of weight tuples of form `(float,float)` corresponding to `(amount,weight)` describing the noise to be added to each link in the graph
- **uniform_amount** (*float*) – the maximum amount of uniform noise to be applied if `noise_weights` is not set

Returns: None

Example

```
>>> from blur.markov.node import Node
>>> node_1 = Node('One')
>>> node_2 = Node('Two')
>>> node_1.add_link(node_1, 3)
>>> node_1.add_link(node_2, 5)
>>> node_2.add_link(node_1, 1)
>>> graph = Graph([node_1, node_2])
>>> for link in graph.node_list[0].link_list:
...     print('{} {}'.format(link.target.value, link.weight))
One 3
```

```
Two 5
>>> graph.apply_noise()
>>> for link in graph.node_list[0].link_list:
...     print('{} {}'.format(
...         link.target.value, link.weight))
One 3.154
Two 5.321
```

find_node_by_value(*value*)

Find and return a node in self.node_list with the value `value`.

If multiple nodes exist with the value `value`, return the first one found.

If no such node exists, this returns `None`.

Parameters **value** (*Any*) – The value of the node to find

Returns

Node – A node with value `value` if it was found

None: If no node exists with value `value`

Example

```
>>> from blur.markov.node import Node
>>> node_1 = Node('One')
>>> graph = Graph([node_1])
>>> found_node = graph.find_node_by_value('One')
>>> found_node == node_1
True
```

remove_node(*node*)

Remove a node from `self.node_list` and links pointing to it.

If `node` is not in the graph, do nothing.

Parameters **node** (`Node`) – The node to be removed

Returns: None

Example

```
>>> from blur.markov.node import Node
>>> node_1 = Node('One')
>>> graph = Graph([node_1])
>>> graph.remove_node(node_1)
>>> len(graph.node_list)
0
```

remove_node_by_value(*value*)

Delete all nodes in `self.node_list` with the value `value`.

Parameters **value** (*Any*) – The value to find and delete owners of.

Returns: None

Example

```
>>> from blur.markov.node import Node
>>> node_1 = Node('One')
>>> graph = Graph([node_1])
>>> graph.remove_node_by_value('One')
>>> len(graph.node_list)
0
```

has_node_with_value(*value*)

Whether any node in `self.node_list` has the value `value`.

Parameters **value** (*Any*) – The value to find in `self.node_list`

Returns: bool

Example

```
>>> from blur.markov.node import Node
>>> node_1 = Node('One')
>>> graph = Graph([node_1])
>>> graph.has_node_with_value('One')
True
>>> graph.has_node_with_value('Foo')
False
```

pick(*starting_node=None*)

Pick a node on the graph based on the links in a starting node.

Additionally, set `self.current_node` to the newly picked node.

- if `starting_node` is specified, start from there
- if `starting_node` is `None`, start from `self.current_node`
- if `starting_node` is `None` and `self.current_node` is `None`, pick a uniformally random node in `self.node_list`

Parameters **starting_node** (`Node`) – `Node` to pick from.

Returns: Node

Example

```
>>> from blur.markov.node import Node
>>> node_1 = Node('One')
>>> node_2 = Node('Two')
>>> node_1.add_link(node_1, 5)
>>> node_1.add_link(node_2, 2)
>>> node_2.add_link(node_1, 1)
>>> graph = Graph([node_1, node_2])
>>> [graph.pick().get_value() for i in range(5)]
['One', 'One', 'Two', 'One', 'One']
```

classmethod **from_string**(*source*, *distance_weights=None*, *merge_same_words=False*, *group_marker_opening='<<'*, *group_marker_closing='>>'*)

Read a string and derive of `Graph` from it.

Words and punctuation marks are made into nodes.

Punctuation marks are split into separate nodes unless they fall between other non-punctuation marks. `'hello,world'` is split into `'hello'`, `','`, and `'world'`, while `'who's there?'` is split into `"who's"`, `'there'`, and `'?'`.

To group arbitrary characters together into a single node (e.g. to make `'hello,world!'`), surround the text in question with `group_marker_opening` and `group_marker_closing`. With the default value, this would look like `'<<hello,world!>>'`. It is recommended that the group markers not appear anywhere in the source text where they aren't meant to act as such to prevent unexpected behavior.

The exact regex for extracting nodes is defined by:

```
expression = r'{0}(.+){1}|([^\w\s]+)\B|([\S]+)'.format(
    ''.join('\' + c for c in group_marker_opening),
    ''.join('\' + c for c in group_marker_closing)
)
```

Parameters

- **source** (*str*) – the string to derive the graph from
- **distance_weights** (*dict*) – dict of relative indices corresponding with word weights. For example, if a dict entry is `1: 1000` this means that every word is linked to the word which follows it with a weight of 1000. `-4: 350` would mean that every word is linked to the 4th word behind it with a weight of 350. A key of `0` refers to the weight words get pointing to themselves. Keys pointing beyond the edge of the word list will wrap around the list.

  The default value for `distance_weights` is `{1: 1}`. This means that each word gets equal weight to whatever word follows it. Consequently, if this default value is used and `merge_same_words` is `False`, the resulting graph behavior will simply move linearly through the source, wrapping at the end to the beginning.
- **merge_same_words** (*bool*) – if nodes which have the same value should be merged or not.
- **group_marker_opening** (*str*) – The string used to mark the beginning of word groups.
- **group_marker_closing** (*str*) – The string used to mark the end of word groups. It is strongly recommended that this be different than `group_marker_opening` to prevent unexpected behavior with the regex pattern.

Returns: Graph

Example

```
>>> graph = Graph.from_string('i have nothing to say and '
...                           'i am saying it and that is poetry.')
>>> ' '.join(graph.pick().value for i in range(8))
'using chance algorithmic in algorithmic art easier blur'
```

classmethod **from_file**(*source*, *distance_weights=None*, *merge_same_words=False*, *group_marker_opening='<<'*, *group_marker_closing='>>'*)

Read a string from a file and derive a `Graph` from it.

This is a convenience function for opening a file and passing its contents to `Graph.from_string()` (see that for more detail)

Parameters

- **source** (*str*) – the file to read and derive the graph from
- **distance_weights** (*dict*) – dict of relative indices corresponding with word weights. See `Graph.from_string` for more detail.
- **merge_same_words** (*bool*) – whether nodes which have the same value should be merged or not.
- **group_marker_opening** (*str*) – The string used to mark the beginning of word groups.
- **group_marker_closing** (*str*) – The string used to mark the end of word groups.

Returns: Graph

Example

```
>>> graph = Graph.from_file('cage.txt')
>>> ' '.join(graph.pick().value for i in range(8))
'poetry i have nothing to say and i'
```

3.3.2 node

Node and Link classes for use in markov graphs

Besides initializing `Node` 's, you will rarely need to directly interact with these objects, as `Graph` provides much easier and more powerful interactions.

class `blur.markov.node.`**Link**(*target*, *weight*)

A one-way link pointing to a `Node` with a weight.

For use in conjunction with the `Node` and `Graph` classes. You will rarely need to deal with `Link` 's directly. The best way to create a `Link` from one `Node` to another is by calling `some_node.add_link(another_node,5)` instead.

Parameters

- **target** (`Node`) – The `Node` this `Link` will point to
- **weight** (*float or int*) – The numerical weight for this `Link`

class `blur.markov.node.`**Node**(*value=None*, *self_destruct=False*)

A node to be used in a Markov graph.

Parameters

- **value** (*Any*) – Value of the node
- **self_destruct** (*bool*) – whether this note deletes itself after being picked by a *Graph*

merge_links_from(*other_node*, *merge_same_value_targets=False*)

Merge links from another node with `self.link_list`.

Copy links from another node, merging when copied links point to a node which this already links to.

Parameters

- **other_node** (Node) – The node to merge links from
- **merge_same_value_targets** (*bool*) – Whether or not to merge links whose targets have the same value (but are not necessarily the same `Node`). If False, links will only be merged when `link_in_other.target == link_in_self.target`. If True, links will be merged when `link_in_other.target.value == link_in_self.target.value`

Returns: None

Example

```
>>> node_1 = Node('One')
>>> node_2 = Node('Two')
>>> node_1.add_link(node_1, 1)
>>> node_1.add_link(node_2, 3)
>>> node_2.add_link(node_1, 4)
>>> node_1.merge_links_from(node_2)
>>> print(node_1)
node.Node instance with value One with 2 links:
    0: 5 --> One
    1: 3 --> Two
```

find_link(*target_node*)

Find the link that points to `target_node` if it exists.

If no link in `self` points to `target_node`, return None

Parameters **target_node** (Node) – The node to look for in `self.link_list`

Returns

Link – An existing link pointing to `target_node` if found

None: If no such link exists

Example

```
>>> node_1 = Node('One')
>>> node_2 = Node('Two')
>>> node_1.add_link(node_2, 1)
>>> link_1 = node_1.link_list[0]
>>> found_link = node_1.find_link(node_2)
>>> found_link == link_1
True
```

add_link(*targets*, *weight*)

Add link(s) pointing to `targets`.

If a link already exists pointing to a target, just add `weight` to that link's weight

Parameters

- **targets** (*Node or list[Node]*) – node or nodes to link to
- **weight** (*int or float*) – weight for the new link(s)

Returns: None

Example

```
>>> node_1 = Node('One')
>>> node_2 = Node('Two')
>>> node_1.add_link(node_2, 1)
>>> new_link = node_1.link_list[0]
>>> print(new_link)
node.Link instance pointing to node with value "Two" with weight 1
```

add_link_to_self(*source*, *weight*)

Create and add a `Link` from a source node to `self`.

Parameters

- **source** (`Node`) – The node that will own the new `Link` pointing to `self`
- **weight** (*int or float*) – The weight of the newly created `Link`

Returns: None

Example

```
>>> node_1 = Node('One')
>>> node_2 = Node('Two')
>>> node_1.add_link_to_self(node_2, 5)
>>> new_link = node_2.link_list[0]
>>> print('{} {}'.format(new_link.target.value, new_link.weight))
One 5
>>> print(new_link)
node.Link instance pointing to node with value "One" with weight 5
```

add_reciprocal_link(*target*, *weight*)

Add links pointing in either direction between `self` and `target`.

This creates a `Link` from `self` to `target` and a `Link` from `target` to `self` of equal weight. If `target` is a list of `Node` 's, repeat this for each one.

Parameters

- **target** (*Node or list[Node]*) –
- **weight** (*int or float*) –

Returns: None

Example

```
>>> node_1 = Node('One')
>>> node_2 = Node('Two')
>>> node_1.add_reciprocal_link(node_2, 5)
>>> new_link_1 = node_1.link_list[0]
```

```
>>> new_link_2 = node_2.link_list[0]
>>> print(new_link_1)
node.Link instance pointing to node with value "Two" with weight 5
>>> print(new_link_2)
node.Link instance pointing to node with value "One" with weight 5
```

remove_links_to_self()

Remove any link in `self.link_list` whose `target` is `self`.

Returns: None

Example

```
>>> node_1 = Node('One')
>>> node_1.add_link(node_1, 5)
>>> node_1.remove_links_to_self()
>>> len(node_1.link_list)
0
```

get_value()

Get the value of this `Node`.

For this class, this simply returns `self.value`, but for subclasses with more complex behavior, this could be more powerful. For example, a `Node` might have a value which is a `SoftColor`, in which case this method could return a `SoftColor.get()` value.

Returns: Any

Example

```
>>> node_1 = Node('One')
>>> node_1.get_value()
'One'
```

3.4 iching

A simple model I Ching.

All data and probabilities taken from Wikipedia at:

- https://en.wikipedia.org/wiki/I_Ching_divination and
- https://en.wikipedia.org/wiki/List_of_hexagrams_of_the_I_Ching

blur.iching.**get_hexagram**(*method='THREE COIN'*)

Return one or two hexagrams using any of a variety of divination methods.

The `NAIVE` method simply returns a uniformally random `int` between `1` and `64`.

All other methods return a 2-tuple where the first value represents the starting hexagram and the second represents the 'moving to' hexagram.

To find the name and unicode glyph for a found hexagram, look it up in the module-level *hexagrams* dict.

Parameters **method** (*str*) – `'THREE COIN'`, `'YARROW'`, or `'NAIVE'`, the divination method model to use. Note that the three coin and yarrow methods are not actually literally simulated, but rather statistical models reflecting the methods are passed to *blur.rand* functions to accurately approximate them.

Returns

int – If `method == 'NAIVE'`, the `int` key of the found hexagram. Otherwise a *tuple* will be returned.

tuple: A 2-tuple of form `(int,int)` where the first value is key of the starting hexagram and the second is that of the 'moving-to' hexagram.

Raises: ValueError if `method` is invalid

Examples:

The function being used alone:

```
>>> get_hexagram(method='THREE COIN')
# Might be...
(55, 2)
>>> get_hexagram(method='YARROW')
# Might be...
(41, 27)
>>> get_hexagram(method='NAIVE')
# Might be...
26
```

Usage in combination with hexagram lookup:

```
>>> grams = get_hexagram()
>>> grams
(47, 42)
# unpack hexagrams for convenient reference
>>> initial, moving_to = grams
>>> hexagrams[initial]
('', '', 'Confining')
>>> hexagrams[moving_to]
('', '', 'Augmenting')
>>> print('{} moving to {}'.format(
...     hexagrams[initial][2],
...     hexagrams[moving_to][2])
...     )
Confining moving to Augmenting
```

`blur.iching.`**`hexagrams`** (*dict*)

A dict of the brief information on the 64 hexagrams in the form

```
{number: (hexagram_symbol, chinese_character, english_translation)}
```

For example,

```
hexagrams = {
    1:  ('', '',   'Force'),
    2:  ('', '',   'Field'),
    3:  ('', '',   'Sprouting'),
    4:  ('', '',   'Enveloping'),
    .
    .
```

```
    .
    64: ('', '',  'Not Yet Fording')
}
```

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

b

W