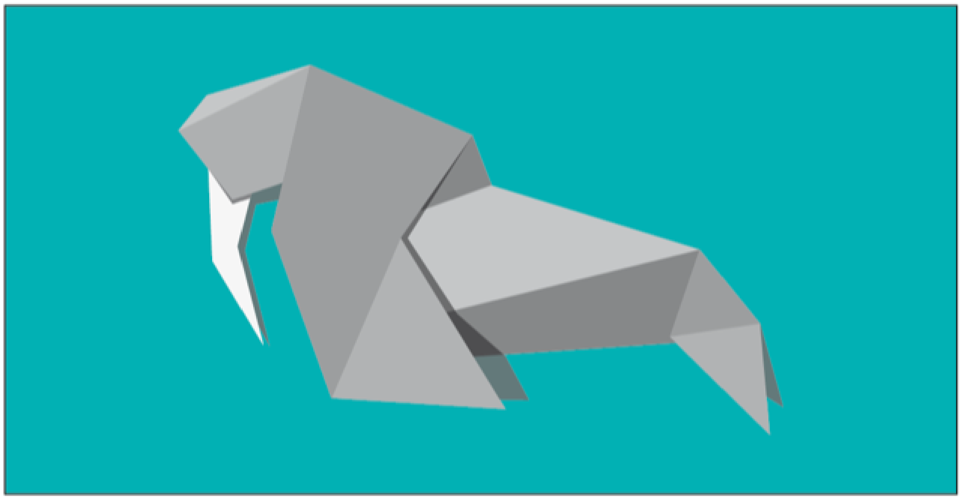
# Whalrus Documentation

*Release 0.4.5*

**François Durand**

**Dec 01, 2020**

# Contents:



CHAPTER 1

# Whalrus

Which Alternative Represents Us, a package for voting rules

- Free software: GNU General Public License v3
- Documentation: https://francois-durand.github.io/whalrus/.

## 1.1 Features

- TODO

## 1.2 Credits

This package was created with Cookiecutter and the audreyr/cookiecutter-pypackage project template.

We use the checklist provided by Package Helper 2.

CHAPTER 2

# Installation

## 2.1 Stable release

To install Whalrus, run this command in your terminal:

```
$ pip install whalrus
```

This is the preferred method to install Whalrus, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

## 2.2 From sources

The sources for Whalrus can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/francois-durand/whalrus
```

Or download the tarball:

```
$ curl  -OL https://github.com/francois-durand/whalrus/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

# CHAPTER 3

# Usage

To use Whalrus in a project:

```
import whalrus
```

# CHAPTER 4

# Tutorial

```
>>> from whalrus import *
```

## 4.1 Quick start

Some simple elections:

```
>>> RulePlurality(['a', 'a', 'b', 'c']).winner_
'a'
>>> RuleBorda(['a > b > c', 'b > c > a']).gross_scores_
{'a': 2, 'b': 3, 'c': 1}
```

Elections can optionally have weights and voter names:

```
>>> RulePlurality(
...     ['a', 'a', 'b', 'c'], weights=[1, 1, 3, 2],
...     voters=['Alice', 'Bob', 'Cate', 'Dave']
... ).winner_
'b'
```

The tie-breaking rule can be specified:

```
>>> RulePlurality(['a', 'a', 'b', 'b', 'c'], tie_break=Priority.ASCENDING).winner_
'a'
```

## 4.2 Computed attributes of an election

```
>>> plurality = RulePlurality(['a', 'a', 'b', 'b', 'c'], tie_break=Priority.
↪ASCENDING)
```

Once the election is defined, you can access its computed attributes, whose names end with an underscore:

```
>>> plurality.candidates_
{'a', 'b', 'c'}
>>> plurality.gross_scores_
{'a': 2, 'b': 2, 'c': 1}
>>> plurality.scores_
{'a': Fraction(2, 5), 'b': Fraction(2, 5), 'c': Fraction(1, 5)}
>>> plurality.best_score_
Fraction(2, 5)
>>> plurality.worst_score_
Fraction(1, 5)
>>> plurality.order_
[{'a', 'b'}, {'c'}]
>>> plurality.strict_order_
['a', 'b', 'c']
>>> plurality.cowinners_
{'a', 'b'}
>>> plurality.winner_
'a'
>>> plurality.cotrailers_
{'c'}
>>> plurality.trailer_
'c'
```

## 4.3 General syntax

In the most general syntax, firstly, you define the rule and enter its options:

```
>>> plurality = RulePlurality(tie_break=Priority.ASCENDING)
```

Secondly, you use it as a callable to load a particular election (profile, set of candidates):

```
>>> plurality(ballots=['a', 'b', 'c'], weights=[2, 2, 1], voters=['Alice', 'Bob',
↪'Cate'],
...           candidates={'a', 'b', 'c', 'd'})  # doctest:+ELLIPSIS
<... object at ...>
```

Finally, you can access the computed variables:

```
>>> plurality.gross_scores_
{'a': 2, 'b': 2, 'c': 1, 'd': 0}
```

Later, if you wish, you can load another profile with the same voting rule, and so on.

## 4.4 Under the hood

A *whalrus.Ballot* contains the message emitted by the voter, but also some contextual information such as the set of candidates that were available at the moment when she cast her ballot:

```
>>> ballot = BallotOrder('a > b ~ c')
>>> ballot
BallotOrder(['a', {'b', 'c'}], candidates={'a', 'b', 'c'})
```

This architecture allows Whalrus to deal with asynchronous elections where the set of candidates may vary during the election itself (such as some asynchronous online polls).

A *whalrus.Profile* contains a list of *whalrus.Ballot* objects, a list of weights and a list of voters:

```
>>> profile = Profile(['a > b ~ c', 'a ~ b > c'])
>>> profile.ballots[0]
BallotOrder(['a', {'b', 'c'}], candidates={'a', 'b', 'c'})
>>> profile.weights
[1, 1]
>>> profile.voters
[None, None]
```

Internally, a voting rule is always applied to a *whalrus.Profile*. Hence, if the inputs are given in a “loose” format, they are converted to a *whalrus.Profile*:

```
>>> borda = RuleBorda(['a > b ~ c', 'a ~ b > c'])
>>> borda.profile_converted_  # doctest:+ELLIPSIS
Profile(ballots=[BallotOrder(['a', {'b', 'c'}], candidates={'a', 'b', 'c'}), ...)
```

Under the hood, some conversions are performed so that a variety of inputs are understood by Whalrus. In the example above, the first ballot was manually entered as `a > b ~ c`. In the absence of other information, Whalrus then considered that only candidates *a*, *b* and *c* were available when this voter cast her ballot. If you want to give more detailed information, the most general syntax consists in using the constructors of classes *whalrus.Profile*, *whalrus.Ballot* and their subclasses:

```
>>> a_more_complex_ballot = BallotOrder('a > b ~ c', candidates={'a', 'b', 'c', 'd',
↪'e'})
```

The ballot above means that the voter emitted the message ‘a > b ~ c’ in a context where the candidates *d* and *e* where also available, i.e. she deliberately abstained about these two candidates.

## 4.5 Change the candidates

It is possible to change the set of candidates, compared to when the voters cast their ballots.

```
>>> profile = Profile(['a > b > c', 'a ~ b > c'])
>>> RulePlurality(profile, candidates={'b', 'c'}).gross_scores_
{'b': 2, 'c': 0}
```

CHAPTER 5

# Reference

## 5.1 Ballot

### 5.1.1 Ballot

**class** whalrus.**Ballot**

A ballot.

The philosophy of this class is to stick as much as possible to the message that the voter emitted, in the context where she emitted it. For example, consider a range voting setting with candidates *a*, *b*, *c* and a scale of grades from 0 to 100. If the voter emits a ballot where *a* has grade 60 and *b* has grade 30, then the `Ballot` object simply records all this: what candidates were present, what was the scale of authorized grades, and what the voter indicated in her ballot. But, for example:

- It makes no assumption whether the voter prefers *a* to *c*. Maybe she did not mention *c* because she didn't like it, maybe because she didn't know it.
- It makes no assumption about what would be the voter's ballot with a scale from 0 to 10. Maybe it would be `{'a': 6, 'b': 3}`, maybe not.

Ballot converters (cf. `ConverterBallot`) will be used each time we need an information that is beyond what the ballot clearly indicated.

**candidates**

The candidates that were available at the moment when the voter cast her ballot. As a consequence, candidates must be hashable objects.

**Type** *NiceSet*

**first** (*candidates: set = None*, ***kwargs*) → object

The first (= most liked) candidate. Implementation is optional.

In most subclasses, this method needs some options (`kwargs`) to solve ambiguities in this conversion. In some other subclasses, this method may even stay unimplemented.

**Parameters**

- **`candidates`** (*`set of candidates`*) – It can be any set of candidates, not necessarily a subset of `self.candidates`). Default: `self.candidates`.
- **`kwargs`** – Some options (depending on the subclass).

**Returns** The first (= most liked) candidate, chosen in the intersection of `self.candidates` and the argument `candidates`. Can return None for an "abstention".

**Return type** candidate

### Examples

Typical example: the ballot was cast in a context where candidates *a*, *b*, *c*, *d* were declared. Hence `self.candidates == {'a', 'b', 'c', 'd'}`. Later, candidate *a* is removed from the election. Then we can use this method with the optional argument `candidates = {'b', 'c', 'd'}` to know who is the most liked candidate of the voter in this new context.

**`last`**(*candidates: set = None*, ***kwargs*) → object

The last (= most disliked) candidate. Implementation is optional.

Cf. *`first()`* for more information.

**Parameters**

- **`candidates`** (*`set of candidates`*) – It can be any set of candidates, not necessarily a subset of `self.candidates`). Default: `self.candidates`.
- **`kwargs`** – Some options (depending on the subclass).

**Returns** The last (= most disliked) candidate, chosen in the intersection of `self.candidates` and the argument `candidates`. Can return None for an "abstention".

**Return type** candidate

**`restrict`**(*candidates=None*, ***kwargs*) → whalrus.ballots.ballot.Ballot

Restrict the ballot to less candidates.

Implementation is optional.

Additional candidates (that are in the argument `candidates` but not in `self.candidates`) are generally not taken into account in the restricted ballot. For example, in a election with candidates *a*, *b*, *c*, assume that the voter emits an ordered ballot `a > b > c`. Later, candidate *a* is removed and candidate *d* is added. Then the "restricted" ballot to `{'b, 'c', 'd'}` is `b > c`. For more details, see for example *`BallotOrder.restrict()`*.

**Parameters**

- **`candidates`** (*`set of candidates`*) – It can be any set of candidates, not necessarily a subset of `self.candidates`). Default: `self.candidates`.
- **`kwargs`** – Some options (depending on the subclass).

**Returns** The same ballot, "restricted" to the candidates given.

**Return type** *Ballot*

## 5.1.2 BallotOrder

**class** `whalrus.`**`BallotOrder`**(*b: object*, *candidates: set = None*)

Ballot with an ordering.

**Parameters**

- **b** (*object*) – The ballot. Cf. examples below for the accepted formats.
- **candidates** (*set*) – The candidates that were available at the moment when the voter cast her ballot. Default: candidates that are explicitly mentioned in the ballot `b`.

### Examples

Most general syntax:

```
>>> ballot = BallotOrder([{'a', 'b'}, {'c'}], candidates={'a', 'b', 'c', 'd', 'e'}
↪)
>>> ballot
BallotOrder([{'a', 'b'}, 'c'], candidates={'a', 'b', 'c', 'd', 'e'})
>>> print(ballot)
a ~ b > c (unordered: d, e)
```

In the example above, candidates *a* and *b* are equally liked, and they are liked better than *c*. Candidates *d* and *e* were available when the voter cast her ballot, but she chose not to include them in her preference order.

Other examples of inputs:

```
>>> BallotOrder('a ~ b > c')
BallotOrder([{'a', 'b'}, 'c'], candidates={'a', 'b', 'c'})
>>> BallotOrder({'a': 10, 'b': 10, 'c': 7})
BallotOrder([{'a', 'b'}, 'c'], candidates={'a', 'b', 'c'})
```

The ballot has a set-like behavior in the sense that it implements `__len__` and `__contains__`:

```
>>> ballot = BallotOrder('a ~ b > c', candidates={'a', 'b', 'c', 'd', 'e'})
>>> len(ballot)
3
>>> 'd' in ballot
False
```

If the order is strict, then the ballot is also iterable:

```
>>> ballot = BallotOrder('a > b > c')
>>> for candidate in ballot:
...     print(candidate)
a
b
c
```

**`as_strict_order`**

Strict order format.

It is a list of candidates. For example, `['a', 'b', 'c']` means that *a* is preferred to *b*, who is preferred to *c*.

**Raises** `ValueError` – If the ballot is not a strict order.

### Examples

```
>>> BallotOrder('a > b > c').as_strict_order
['a', 'b', 'c']
```

**Type** list

**as_weak_order**

Weak order format.

A list of sets. For example, `[{'a', 'b'}, {'c'}]` means that *a* and *b* are equally liked, and they are liked better than *c*.

### Examples

```
>>> BallotOrder('a ~ b > c', candidates={'a', 'b', 'c', 'd', 'e'}).as_weak_
↪order
[{'a', 'b'}, {'c'}]
```

**Type** list

**candidates**

the candidates.

If the set was not explicitly given, the candidates are inferred from the ballot.

### Examples

```
>>> BallotOrder('a ~ b > c', candidates={'a', 'b', 'c', 'd', 'e'}).candidates
{'a', 'b', 'c', 'd', 'e'}
>>> BallotOrder('a ~ b > c').candidates
{'a', 'b', 'c'}
```

**Type** *NiceSet*

**candidates_in_b**

the candidates that are explicitly mentioned in the ballot.

### Examples

```
>>> BallotOrder('a ~ b > c', candidates={'a', 'b', 'c', 'd', 'e'}).candidates_
↪in_b
{'a', 'b', 'c'}
```

**Type** *NiceSet*

**candidates_not_in_b**

the candidates that were available at the moment of the vote, but are not explicitly mentioned in the ballot.

### Examples

```
>>> BallotOrder('a ~ b > c', candidates={'a', 'b', 'c', 'd', 'e'}).candidates_
↪not_in_b
{'d', 'e'}
```

**Type** *NiceSet*

**first** (*candidates: set = None*, ***kwargs*) → object
The first (= most liked) candidate.

**Parameters**

- **candidates** (*set of candidates*) – It can be any set of candidates, not necessarily a subset of `self.candidates`. Default: `self.candidates`.
- **kwargs** –
  - *priority*: a *Priority*. Default: *Priority.UNAMBIGUOUS*.
  - *include_unordered*: a boolean. If True (default), then unordered candidates are considered present but below the others.

**Returns** The first (= most liked) candidate, chosen in the intersection of `self.candidates` and the argument `candidates`. Can return None for an "abstention".

**Return type** candidate

### Examples

```
>>> print(BallotOrder('a ~ b').first(priority=Priority.ASCENDING))
a
>>> print(BallotOrder('a > b', candidates={'a', 'b', 'c'}).first(candidates={
↪'c'}))
c
>>> print(BallotOrder('a > b', candidates={'a', 'b', 'c'}).first(candidates={
↪'c'},
...                                                                 include_
↪unordered=False))
None
```

**is_strict**
Whether the ballot is a strict order or not.

True if the order is strict, i.e. if each indifference class contains one element. There can be some unordered candidates.

### Examples

```
>>> BallotOrder('a > b > c').is_strict
True
>>> BallotOrder('a > b > c', candidates={'a', 'b', 'c', 'd', 'e'}).is_strict
True
>>> BallotOrder('a ~ b > c').is_strict
False
```

**Type** bool

**last** (*candidates: set = None*, ***kwargs*) → object
The last (= most disliked) candidate.

**Parameters**

- **candidates** (*set of candidates*) – It can be any set of candidates, not necessarily a subset of `self.candidates`. Default is `self.candidates`.

- **kwargs** –
  - *priority*: a *Priority* object. Default: *Priority.UNAMBIGUOUS*.
  - *include_unordered*: a boolean. If True (default), then unordered candidates are considered present but below the others.

**Returns** The last (= most disliked) candidate, chosen in the intersection of `self.candidates` and the argument `candidates`. Can return None for an "abstention".

**Return type** candidate

### Examples

```
>>> print(BallotOrder('a ~ b').last(priority=Priority.ASCENDING))
b
>>> print(BallotOrder('a > b', candidates={'a', 'b', 'c'}).last())
c
>>> print(BallotOrder('a > b', candidates={'a', 'b', 'c'}).last(include_
↪unordered=False))
b
>>> ballot = BallotOrder('a > b', candidates={'a', 'b', 'c', 'd'})
>>> print(ballot.last(candidates={'c', 'd'}, include_unordered=False))
None
```

**restrict**(*candidates: set = None*, ***kwargs*) → whalrus.ballots.ballot_order.BallotOrder

Restrict the ballot to less candidates.

**Parameters**

- **candidates** (*set of candidates*) – It can be any set of candidates, not necessarily a subset of `self.candidates`). Default: `self.candidates`.
- **kwargs** – Some options (depending on the subclass).

**Returns** The same ballot, "restricted" to the candidates given.

**Return type** *BallotOrder*

### Examples

Typical usage:

```
>>> ballot = BallotOrder('a ~ b > c')
>>> ballot
BallotOrder([{'a', 'b'}, 'c'], candidates={'a', 'b', 'c'})
>>> ballot.restrict(candidates={'b', 'c'})
BallotOrder(['b', 'c'], candidates={'b', 'c'})
```

More general usage:

```
>>> ballot.restrict(candidates={'b', 'c', 'd'})
BallotOrder(['b', 'c'], candidates={'b', 'c'})
```

In the last example above, note that *d* is not in the candidates of the restricted ballot, as she was not available at the moment when the voter cast her ballot.

### 5.1.3 BallotLevels

**class** whalrus.**BallotLevels**(*b: dict*, *candidates: set = None*, *scale: whalrus.scales.scale.Scale = None*)

Ballot with an evaluation of the candidates.

**Parameters**

- **b** (*dict*) – Keys: candidates. Values represent some form of evaluation. The keys and the values must be hashable.
- **candidates** (*set*) – The candidates that were available at the moment when the voter cast her ballot. Default: candidates that are explicitly mentioned in the ballot `b`.
- **scale** (*Scale*) – The authorized scale of evaluations at the moment when the voter cast her ballot. Default: `Scale()` (meaning in this case "unknown").

#### Examples

Most general syntax:

```
>>> ballot = BallotLevels({'a': 10, 'b': 7, 'c': 3},
...                       candidates={'a', 'b', 'c', 'd', 'e'},
...                       scale=ScaleRange(low=0, high=10))
```

Other examples of syntax:

```
>>> ballot = BallotLevels({'a': 10, 'b': 7, 'c': 3})
>>> ballot = BallotLevels({'a': 'Good', 'b': 'Bad', 'c': 'Bad'},
...                       scale=ScaleFromList(['Bad', 'Medium', 'Good']))
```

In addition to the set-like and list-like behaviors defined in parent class *BallotOrder*, it also has a dictionary-like behavior in the sense that it implements `__getitem__`:

```
>>> ballot = BallotLevels({'a': 10, 'b': 7, 'c': 3})
>>> ballot['a']
10
```

**as_dict**

keys are candidates and values are levels of evaluation.

#### Examples

```
>>> BallotLevels({'a': 10, 'b': 7, 'c': 3}).as_dict
{'a': 10, 'b': 7, 'c': 3}
```

**Type** *NiceDict*

**as_strict_order**

Strict order format.

It is a list of candidates. For example, `['a', 'b', 'c']` means that *a* is preferred to *b*, who is preferred to *c*.

**Raises** `ValueError` – If the ballot is not a strict order.

### Examples

```
>>> BallotOrder('a > b > c').as_strict_order
['a', 'b', 'c']
```

**Type** list

**candidates**

the candidates.

If the set was not explicitly given, the candidates are inferred from the ballot.

### Examples

```
>>> BallotOrder('a ~ b > c', candidates={'a', 'b', 'c', 'd', 'e'}).candidates
{'a', 'b', 'c', 'd', 'e'}
>>> BallotOrder('a ~ b > c').candidates
{'a', 'b', 'c'}
```

**Type** *NiceSet*

**candidates_not_in_b**

the candidates that were available at the moment of the vote, but are not explicitly mentioned in the ballot.

### Examples

```
>>> BallotOrder('a ~ b > c', candidates={'a', 'b', 'c', 'd', 'e'}).candidates_
↪not_in_b
{'d', 'e'}
```

**Type** *NiceSet*

**first** (*candidates: set = None*, ***kwargs*) → object

The first (= most liked) candidate.

**Parameters**

- **candidates** (`set of candidates`) – It can be any set of candidates, not necessarily a subset of `self.candidates`. Default: `self.candidates`.
- **kwargs** –
  - *priority*: a `Priority`. Default: `Priority.UNAMBIGUOUS`.
  - *include_unordered*: a boolean. If True (default), then unordered candidates are considered present but below the others.

**Returns** The first (= most liked) candidate, chosen in the intersection of `self.candidates` and the argument `candidates`. Can return None for an "abstention".

**Return type** candidate

### Examples

```
>>> print(BallotOrder('a ~ b').first(priority=Priority.ASCENDING))
a
>>> print(BallotOrder('a > b', candidates={'a', 'b', 'c'}).first(candidates={
↪'c'}))
c
>>> print(BallotOrder('a > b', candidates={'a', 'b', 'c'}).first(candidates={
↪'c'},
...                                                                 include_
↪unordered=False))
None
```

**is_strict**

Whether the ballot is a strict order or not.

True if the order is strict, i.e. if each indifference class contains one element. There can be some unordered candidates.

### Examples

```
>>> BallotOrder('a > b > c').is_strict
True
>>> BallotOrder('a > b > c', candidates={'a', 'b', 'c', 'd', 'e'}).is_strict
True
>>> BallotOrder('a ~ b > c').is_strict
False
```

**Type** bool

**items**() → ItemsView[KT, VT_co]

Items of the ballot.

**Returns** This is a shortcut for `self.as_dict.items()`.

**Return type** ItemsView

### Examples

```
>>> ballot = BallotLevels({'a': 10, 'b': 7, 'c': 3}, candidates={'a', 'b', 'c
↪', 'd', 'e'})
>>> sorted(ballot.items())
[('a', 10), ('b', 7), ('c', 3)]
```

**keys**() → KeysView[KT]

Keys of the ballot.

**Returns** This is a shortcut for `self.as_dict.keys()`.

**Return type** KeysView

### Examples

```
>>> ballot = BallotLevels({'a': 10, 'b': 7, 'c': 3}, candidates={'a', 'b', 'c
↪', 'd', 'e'})
>>> sorted(ballot.keys())
['a', 'b', 'c']
```

**last** (*candidates: set = None*, ***kwargs*) → object

The last (= most disliked) candidate.

**Parameters**

- **candidates** (*set of candidates*) – It can be any set of candidates, not necessarily a subset of `self.candidates`. Default is `self.candidates`.
- **kwargs** –
  - *priority*: a *Priority* object. Default: *Priority.UNAMBIGUOUS*.
  - *include_unordered*: a boolean. If True (default), then unordered candidates are considered present but below the others.

**Returns** The last (= most disliked) candidate, chosen in the intersection of `self.candidates` and the argument `candidates`. Can return None for an "abstention".

**Return type** candidate

### Examples

```
>>> print(BallotOrder('a ~ b').last(priority=Priority.ASCENDING))
b
>>> print(BallotOrder('a > b', candidates={'a', 'b', 'c'}).last())
c
>>> print(BallotOrder('a > b', candidates={'a', 'b', 'c'}).last(include_
↪unordered=False))
b
>>> ballot = BallotOrder('a > b', candidates={'a', 'b', 'c', 'd'})
>>> print(ballot.last(candidates={'c', 'd'}, include_unordered=False))
None
```

**restrict** (*candidates: set = None*, ***kwargs*) → whalrus.ballots.ballot_levels.BallotLevels

Restrict the ballot to less candidates.

**Parameters**

- **candidates** (*set of candidates*) – It can be any set of candidates, not necessarily a subset of `self.candidates`). Default: `self.candidates`.
- **kwargs** – Some options (depending on the subclass).

**Returns** The same ballot, "restricted" to the candidates given.

**Return type** *BallotOrder*

### Examples

Typical usage:

```
>>> ballot = BallotOrder('a ~ b > c')
>>> ballot
BallotOrder([{'a', 'b'}, 'c'], candidates={'a', 'b', 'c'})
>>> ballot.restrict(candidates={'b', 'c'})
BallotOrder(['b', 'c'], candidates={'b', 'c'})
```

More general usage:

```
>>> ballot.restrict(candidates={'b', 'c', 'd'})
BallotOrder(['b', 'c'], candidates={'b', 'c'})
```

In the last example above, note that *d* is not in the candidates of the restricted ballot, as she was not available at the moment when the voter cast her ballot.

**values**() → ValuesView[VT_co]

Values of the ballot.

**Returns** This is a shortcut for `self.as_dict.values()`.

**Return type** ValuesView

### Examples

```
>>> ballot = BallotLevels({'a': 10, 'b': 7, 'c': 3}, candidates={'a', 'b', 'c
↪', 'd', 'e'})
>>> sorted(ballot.values())
[3, 7, 10]
```

## 5.1.4 BallotOneName

**class** whalrus.**BallotOneName**(*b: object*, *candidates: set = None*)

A ballot in a mono-nominal context (typically plurality or veto).

**Parameters**

- **b** (*candidate or None*) – None stands for abstention.
- **candidates** (*set*) – The candidates that were available at the moment when the voter cast her ballot.

### Examples

```
>>> ballot = BallotOneName('a', candidates={'a', 'b', 'c'})
>>> print(ballot)
a
```

```
>>> ballot = BallotOneName(None, candidates={'a', 'b', 'c'})
>>> print(ballot)
None
```

**candidates_in_b**

The candidate that is explicitly mentioned in the ballot.

This is a singleton with the only candidate contained in the ballot (or an empty set in case of abstention).

### Examples

```
>>> BallotOneName('a', candidates={'a', 'b', 'c'}).candidates_in_b
{'a'}
>>> BallotOneName(None, candidates={'a', 'b', 'c'}).candidates_in_b
{}
```

**Type** *NiceSet*

**`candidates_not_in_b`**
The candidates that were available at the moment of the vote, but are not explicitly mentioned in the ballot.

### Examples

```
>>> BallotOneName('a', candidates={'a', 'b', 'c'}).candidates_not_in_b
{'b', 'c'}
```

**Type** *NiceSet*

**`first`** (*candidates: set = None*, ***kwargs*) → object
The first (= most liked) candidate.

In this parent class, by default, the ballot is considered as a plurality ballot, i.e. the candidate indicated is the most liked.

**Parameters**

- **candidates** (*set of candidates*) –
- **kwargs** –
  - *priority*: a `Priority`. Default: `Priority.UNAMBIGUOUS`.

**Returns** The first (= most liked) candidate.

**Return type** candidate

### Examples

```
>>> BallotOneName('a', candidates={'a', 'b', 'c'}).first()
'a'
>>> BallotOneName('a', candidates={'a', 'b', 'c'}).first(candidates={'b', 'c'}
↪,
...                                                         priority=Priority.
↪ASCENDING)
'b'
```

**`last`** (*candidates: set = None*, ***kwargs*) → object
The last (= most disliked) candidate.

In this parent class, by default, the ballot is considered as a plurality ballot, i.e. the candidate indicated is the most liked.

**Parameters**

- **candidates** (*set of candidates*) –

- **kwargs** –
  - *priority*: a `Priority`. Default: `Priority.UNAMBIGUOUS`.

**Returns** The last (= most disliked) candidate.

**Return type** candidate

#### Examples

```
>>> BallotOneName('a', candidates={'a', 'b'}).last()
'b'
>>> BallotOneName('a', candidates={'a', 'b', 'c'}).last(priority=Priority.
↪ASCENDING)
'c'
```

**restrict** (*candidates: set = None*, ***kwargs*) → whalrus.ballots.ballot_one_name.BallotOneName
Restrict the ballot to less candidates.

**Parameters**

- **candidates** (`set of candidates`) – It can be any set of candidates, not necessarily a subset of `self.candidates`). Default: `self.candidates`.
- **kwargs** –
  - *priority*: a `Priority`. Default: `Priority.UNAMBIGUOUS`.

**Returns** The same ballot, “restricted” to the candidates given.

**Return type** *BallotOneName*

#### Examples

```
>>> BallotOneName('a', candidates={'a', 'b'}).restrict(candidates={'b'})
BallotOneName('b', candidates={'b'})
>>> BallotOneName('a', candidates={'a', 'b', 'c'}).restrict(candidates={'b',
↪'c'},
...                                                        priority=Priority.
↪ASCENDING)
BallotOneName('b', candidates={'b', 'c'})
```

### 5.1.5 BallotPlurality

**class** whalrus.**BallotPlurality** (*b: object*, *candidates: set = None*)
A plurality ballot.

#### Examples

```
>>> ballot = BallotPlurality('a', candidates={'a', 'b', 'c'})
>>> print(ballot)
a
```

```
>>> ballot = BallotPlurality(None, candidates={'a', 'b', 'c'})
>>> print(ballot)
None
```

**candidates_in_b**

The candidate that is explicitly mentioned in the ballot.

This is a singleton with the only candidate contained in the ballot (or an empty set in case of abstention).

### Examples

```
>>> BallotOneName('a', candidates={'a', 'b', 'c'}).candidates_in_b
{'a'}
>>> BallotOneName(None, candidates={'a', 'b', 'c'}).candidates_in_b
{}
```

**Type** *NiceSet*

**candidates_not_in_b**

The candidates that were available at the moment of the vote, but are not explicitly mentioned in the ballot.

### Examples

```
>>> BallotOneName('a', candidates={'a', 'b', 'c'}).candidates_not_in_b
{'b', 'c'}
```

**Type** *NiceSet*

**first**(*candidates: set = None*, ***kwargs*) → object

The first (= most liked) candidate.

In this parent class, by default, the ballot is considered as a plurality ballot, i.e. the candidate indicated is the most liked.

**Parameters**

- **candidates** (*set of candidates*) –
- **kwargs** –
  - *priority*: a *Priority*. Default: *Priority.UNAMBIGUOUS*.

**Returns** The first (= most liked) candidate.

**Return type** candidate

### Examples

```
>>> BallotOneName('a', candidates={'a', 'b', 'c'}).first()
'a'
>>> BallotOneName('a', candidates={'a', 'b', 'c'}).first(candidates={'b', 'c'}
↪,
```

(continues on next page)

(continued from previous page)

```
...                                                         priority=Priority.
↪ASCENDING)
'b'
```

**last** (*candidates: set = None*, ***kwargs*) → object

The last (= most disliked) candidate.

In this parent class, by default, the ballot is considered as a plurality ballot, i.e. the candidate indicated is the most liked.

**Parameters**

- **candidates** (*set of candidates*) –
- **kwargs** –
  - *priority*: a *Priority*. Default: *Priority.UNAMBIGUOUS*.

**Returns** The last (= most disliked) candidate.

**Return type** candidate

### Examples

```
>>> BallotOneName('a', candidates={'a', 'b'}).last()
'b'
>>> BallotOneName('a', candidates={'a', 'b', 'c'}).last(priority=Priority.
↪ASCENDING)
'c'
```

**restrict** (*candidates: set = None*, ***kwargs*) → whalrus.ballots.ballot_one_name.BallotOneName

Restrict the ballot to less candidates.

**Parameters**

- **candidates** (*set of candidates*) – It can be any set of candidates, not necessarily a subset of `self.candidates`). Default: `self.candidates`.
- **kwargs** –
  - *priority*: a *Priority*. Default: *Priority.UNAMBIGUOUS*.

**Returns** The same ballot, “restricted” to the candidates given.

**Return type** *BallotOneName*

### Examples

```
>>> BallotOneName('a', candidates={'a', 'b'}).restrict(candidates={'b'})
BallotOneName('b', candidates={'b'})
>>> BallotOneName('a', candidates={'a', 'b', 'c'}).restrict(candidates={'b',
↪'c'},
...                                                               priority=Priority.
↪ASCENDING)
BallotOneName('b', candidates={'b', 'c'})
```

### 5.1.6 BallotVeto

**class** whalrus.**BallotVeto**(*b: object*, *candidates: set = None*)

A veto (anti-plurality) ballot.

#### Examples

```
>>> ballot = BallotVeto('a', candidates={'a', 'b', 'c'})
>>> print(ballot)
a
```

```
>>> ballot = BallotVeto(None, candidates={'a', 'b', 'c'})
>>> print(ballot)
None
```

**candidates_in_b**

The candidate that is explicitly mentioned in the ballot.

This is a singleton with the only candidate contained in the ballot (or an empty set in case of abstention).

#### Examples

```
>>> BallotOneName('a', candidates={'a', 'b', 'c'}).candidates_in_b
{'a'}
>>> BallotOneName(None, candidates={'a', 'b', 'c'}).candidates_in_b
{}
```

**Type** *NiceSet*

**candidates_not_in_b**

The candidates that were available at the moment of the vote, but are not explicitly mentioned in the ballot.

#### Examples

```
>>> BallotOneName('a', candidates={'a', 'b', 'c'}).candidates_not_in_b
{'b', 'c'}
```

**Type** *NiceSet*

**first**(*candidates: set = None*, ***kwargs*) → object

#### Examples

```
>>> BallotVeto('a', candidates={'a', 'b'}).first()
'b'
>>> BallotVeto('a', candidates={'a', 'b', 'c'}).first(priority=Priority.
↪ASCENDING)
'b'
```

**last**(*candidates: set = None*, ***kwargs*) → object

### Examples

```
>>> BallotVeto('a', candidates={'a', 'b', 'c'}).last()
'a'
>>> BallotVeto('a', candidates={'a', 'b', 'c'}).last(candidates={'b', 'c'},
...                                                   priority=Priority.
↪ASCENDING)
'c'
```

**restrict** (*candidates: set = None*, ***kwargs*) → whalrus.ballots.ballot_one_name.BallotOneName

Restrict the ballot to less candidates.

**Parameters**

- **candidates** (*set of candidates*) – It can be any set of candidates, not necessarily a subset of `self.candidates`). Default: `self.candidates`.
- **kwargs** –
  - *priority*: a `Priority`. Default: `Priority.UNAMBIGUOUS`.

**Returns** The same ballot, "restricted" to the candidates given.

**Return type** *BallotOneName*

### Examples

```
>>> BallotOneName('a', candidates={'a', 'b'}).restrict(candidates={'b'})
BallotOneName('b', candidates={'b'})
>>> BallotOneName('a', candidates={'a', 'b', 'c'}).restrict(candidates={'b',
↪'c'},
...                                                          priority=Priority.
↪ASCENDING)
BallotOneName('b', candidates={'b', 'c'})
```

## 5.2 ConverterBallot

### 5.2.1 ConverterBallot

**class** whalrus.**ConverterBallot**

A ballot converter.

A converter is a callable. Its input may have various formats. Its output must be a `Ballot`, often of a specific subclass. For more information and examples, cf. `ConverterBallotGeneral`.

### 5.2.2 ConverterBallotGeneral

**class** whalrus.**ConverterBallotGeneral** (*plurality_priority: whalrus.priorities.priority.Priority = Priority.UNAMBIGUOUS*, *veto_priority: whalrus.priorities.priority.Priority = Priority.UNAMBIGUOUS*, *one_name_priority: whalrus.priorities.priority.Priority = Priority.UNAMBIGUOUS*)

General ballot converter.

This is a default general converter. It tries to infer the type of input and converts it to an object of the relevant subclass of `Ballot`.

**Parameters**

- **plurality_priority** (`Priority`) – Option passed to `BallotPlurality.restrict()` when restricting the ballot if, once converted, it is a `BallotPlurality`.
- **veto_priority** (`Priority`) – Option passed to `BallotVeto.restrict()` when restricting the ballot if, once converted, if is a `BallotVeto`.
- **one_name_priority** (`Priority`) – Option passed to `BallotOneName.restrict()` when restricting the ballot if, once converted, it is a `BallotOneName` (but not a `BallotPlurality` or `BallotVeto`).

### Examples

Typical usage:

```
>>> converter = ConverterBallotGeneral()
>>> converter({'a': 10, 'b': 7, 'c': 0})
BallotLevels({'a': 10, 'b': 7, 'c': 0}, candidates={'a', 'b', 'c'}, scale=Scale())
>>> converter([{'a', 'b'}, {'c'}])
BallotOrder([{'a', 'b'}, 'c'], candidates={'a', 'b', 'c'})
>>> converter('a ~ b > c')
BallotOrder([{'a', 'b'}, 'c'], candidates={'a', 'b', 'c'})
>>> converter('Alice')
BallotOneName('Alice', candidates={'Alice'})
```

It is also possible to “restrict” the set of candidates on-the-fly:

```
>>> converter = ConverterBallotGeneral()
>>> converter('a ~ b > c', candidates={'b', 'c'})
BallotOrder(['b', 'c'], candidates={'b', 'c'})
>>> converter({'a': 10, 'b': 7, 'c': 0}, candidates={'b', 'c'})
BallotLevels({'b': 7, 'c': 0}, candidates={'b', 'c'}, scale=Scale())
```

Cf. `Ballot.restrict()` for more information.

Use options for the restrictions:

```
>>> converter = ConverterBallotGeneral(one_name_priority=Priority.ASCENDING,
...                                    plurality_priority=Priority.ASCENDING,
...                                    veto_priority=Priority.ASCENDING)
>>> converter(BallotOneName('a', candidates={'a', 'b', 'c'}), candidates={'b', 'c
↪'})
BallotOneName('b', candidates={'b', 'c'})
>>> converter(BallotPlurality('a', candidates={'a', 'b', 'c'}), candidates={'b',
↪'c'})
BallotPlurality('b', candidates={'b', 'c'})
>>> converter(BallotVeto('a', candidates={'a', 'b', 'c'}), candidates={'b', 'c'})
BallotVeto('c', candidates={'b', 'c'})
```

## 5.2.3 ConverterBallotToGrades

**class** whalrus.**ConverterBallotToGrades**(*scale: whalrus.scales.scale.Scale = None, borda_unordered_give_points: bool = True*)

Default converter to a `BallotLevels` using numeric grades.

This is a default converter to a `BallotLevels` using numeric grades. It tries to infer the type of input and converts it to a `BallotLevels`, with a numeric scale. It is a wrapper for the specialized converters `ConverterBallotToLevelsInterval`, `ConverterBallotToLevelsRange`, and `ConverterBallotToLevelsListNumeric`.

**Parameters**

- **scale** (numeric `Scale`) – If specified, then the ballot will be converted to this scale. If it is None, then any ballot that is of class `BallotLevels` and numeric will be kept as it is, and any other ballot will converted to a `BallotLevels` using a `ScaleInterval` with bounds 0 and 1.
- **borda_unordered_give_points** (*bool*) – When converting a `BallotOrder` that is not a `BallotLevels`, we use Borda scores as a calculation step. This parameter decides whether the unordered candidates of the ballot give points to the ordered candidates. Cf. `ScorerBorda`.

### Examples

Typical usages:

```
>>> ballot = BallotLevels({'a': 100, 'b': 57}, scale=ScaleRange(0, 100))
>>> ConverterBallotToGrades(scale=ScaleInterval(low=0, high=10))(ballot).as_dict
{'a': 10, 'b': Fraction(57, 10)}
>>> ConverterBallotToGrades(scale=ScaleRange(low=0, high=10))(ballot).as_dict
{'a': 10, 'b': 6}
>>> ConverterBallotToGrades(scale=ScaleFromSet({0, 2, 4, 10}))(ballot).as_dict
{'a': 10, 'b': 4}
```

```
>>> ballot = BallotLevels({'a': 'Good', 'b': 'Medium'},
...                       scale=ScaleFromList(['Bad', 'Medium', 'Good']))
>>> ConverterBallotToGrades()(ballot).as_dict
{'a': 1, 'b': Fraction(1, 2)}
```

For more examples, cf. `ConverterBallotToLevelsInterval`, `ConverterBallotToLevelsRange`, and `ConverterBallotToLevelsListNumeric`.

## 5.2.4 ConverterBallotToLevels

**class** whalrus.**ConverterBallotToLevels**(*scale: whalrus.scales.scale.Scale = None, borda_unordered_give_points: bool = True*)

Default converter to a `BallotLevels` (representing grades, appreciations, etc).

This is a default converter to a `BallotLevels`. It tries to infer the type of input and converts it to a `BallotLevels`. It is a wrapper for the specialized converters `ConverterBallotToLevelsInterval`, `ConverterBallotToLevelsRange`, `ConverterBallotToLevelsListNumeric`, and `ConverterBallotToLevelsListNonNumeric`.

**Parameters**

- **scale** (`Scale`) – If specified, then the ballot will be converted to this scale. If it is None, then any ballot of class `BallotLevels` will be kept as it is, and any other ballot will converted to a `BallotLevels` using a `ScaleInterval` with bounds 0 and 1
- **borda_unordered_give_points** (*bool*) – When converting a `BallotOrder` that is not a `BallotLevels`, we use Borda scores as a calculation step. This parameter decides

whether the unordered candidates of the ballot give points to the ordered candidates. Cf. *ScorerBorda*.

### Examples

Typical usages:

```
>>> ballot = BallotLevels({'a': 100, 'b': 57}, scale=ScaleRange(0, 100))
>>> ConverterBallotToLevels(scale=ScaleInterval(low=0, high=10))(ballot).as_dict
{'a': 10, 'b': Fraction(57, 10)}
>>> ConverterBallotToLevels(scale=ScaleRange(low=0, high=10))(ballot).as_dict
{'a': 10, 'b': 6}
>>> ConverterBallotToLevels(scale=ScaleFromList([
...     'Bad', 'Medium', 'Good', 'Very Good', 'Great', 'Excellent']))(ballot).as_
↪dict
{'a': 'Excellent', 'b': 'Very Good'}
>>> ConverterBallotToLevels(scale=ScaleFromSet({0, 2, 4, 10}))(ballot).as_dict
{'a': 10, 'b': 4}
```

For more examples, cf. *ConverterBallotToLevelsInterval*, *ConverterBallotToLevelsRange*, *ConverterBallotToLevelsListNumeric*, and *ConverterBallotToLevelsListNonNumeric*.

## 5.2.5 ConverterBallotToLevelsInterval

**class** whalrus.**ConverterBallotToLevelsInterval**(*scale: whalrus.scales.scale.Scale = ScaleInterval(low=0, high=1), borda_unordered_give_points: bool = True*)

0 Default converter to a *BallotLevels* using a *ScaleInterval* (interval of real numbers).

**Parameters**

- **scale** (ScaleInterval) –
- **borda_unordered_give_points** (*bool*) – When converting a *BallotOrder* that is not a *BallotLevels*, we use Borda scores (normalized to the interval [scale.low, scale.high]. This parameter decides whether the unordered candidates of the ballot give points to the ordered candidates. Cf. *ScorerBorda*.

### Examples

Typical usages:

```
>>> converter = ConverterBallotToLevelsInterval()
>>> b = BallotLevels({'a': 1, 'b': .5}, candidates={'a', 'b', 'c'},␣
↪scale=ScaleInterval(-1, 1))
>>> converter(b).as_dict
{'a': 1, 'b': Fraction(3, 4)}
>>> b = BallotLevels({'a': 5, 'b': 4}, candidates={'a', 'b', 'c'},␣
↪scale=ScaleRange(0, 5))
>>> converter(b).as_dict
{'a': 1, 'b': Fraction(4, 5)}
>>> b = BallotLevels({'a': 3, 'b': 0}, candidates={'a', 'b', 'c'},␣
↪scale=ScaleFromSet({-1, 0, 3}))
```

(continues on next page)

(continued from previous page)

```
>>> converter(b).as_dict
{'a': 1, 'b': Fraction(1, 4)}
>>> b = BallotLevels({'a': 'Excellent', 'b': 'Very Good'}, candidates={'a', 'b',
↪'c'},
...                   scale=ScaleFromList(['Bad', 'Medium', 'Good', 'Very Good',
↪'Excellent']))
>>> converter(b).as_dict
{'a': 1, 'b': Fraction(3, 4)}
>>> converter(BallotOneName('a', candidates={'a', 'b', 'c'})).as_dict
{'a': 1, 'b': 0, 'c': 0}
>>> converter(BallotPlurality('a', candidates={'a', 'b', 'c'})).as_dict
{'a': 1, 'b': 0, 'c': 0}
>>> converter(BallotVeto('a', candidates={'a', 'b', 'c'})).as_dict
{'a': 0, 'b': 1, 'c': 1}
>>> converter('a > b > c').as_dict
{'a': 1, 'b': Fraction(1, 2), 'c': 0}
```

Options for converting ordered ballots:

```
>>> b = BallotOrder('a > b > c', candidates={'a', 'b', 'c', 'd', 'e'})
>>> ConverterBallotToLevelsInterval(borda_unordered_give_points=False)(b).as_dict
{'a': 1, 'b': Fraction(1, 2), 'c': 0}
>>> ConverterBallotToLevelsInterval(borda_unordered_give_points=True)(b).as_dict
{'a': 1, 'b': Fraction(3, 4), 'c': Fraction(1, 2)}
```

### 5.2.6 ConverterBallotToLevelsListNonNumeric

**class** whalrus.**ConverterBallotToLevelsListNonNumeric**(*scale: whalrus.scales.scale_from_list.ScaleFromList*, *borda_unordered_give_points: bool = True*)

Default converter to a *BallotLevels* using a *ScaleFromList* of levels that are not numbers.

This converter works essentially the same as *ConverterBallotToLevelsInterval*, but it then maps the evaluation to levels of the scale.

**Parameters**

- **scale** (ScaleFromList) – The scale.
- **borda_unordered_give_points** (*bool*) – When converting a *BallotOrder* that is not a *BallotLevels*, we use Borda scores as a calculation step. This parameter decides whether the unordered candidates of the ballot give points to the ordered candidates. Cf. *ScorerBorda*.

#### Examples

Typical usages:

```
>>> converter = ConverterBallotToLevelsListNonNumeric(
...     scale=ScaleFromList(['Bad', 'Medium', 'Good', 'Very Good', 'Great',
↪'Excellent']))
>>> b = BallotLevels({'a': 1, 'b': .2}, candidates={'a', 'b', 'c'},␣
↪scale=ScaleInterval(-1, 1))
>>> converter(b).as_dict
```

(continues on next page)

(continued from previous page)

```
{'a': 'Excellent', 'b': 'Very Good'}
>>> b = BallotLevels({'a': 5, 'b': 4}, candidates={'a', 'b', 'c'},␣
↪scale=ScaleRange(0, 5))
>>> converter(b).as_dict
{'a': 'Excellent', 'b': 'Great'}
>>> b = BallotLevels({'a': 4, 'b': 0}, candidates={'a', 'b', 'c'},␣
↪scale=ScaleFromSet({-1, 0, 4}))
>>> converter(b).as_dict
{'a': 'Excellent', 'b': 'Medium'}
>>> converter(BallotOneName('a', candidates={'a', 'b', 'c'})).as_dict
{'a': 'Excellent', 'b': 'Bad', 'c': 'Bad'}
>>> converter(BallotPlurality('a', candidates={'a', 'b', 'c'})).as_dict
{'a': 'Excellent', 'b': 'Bad', 'c': 'Bad'}
>>> converter(BallotVeto('a', candidates={'a', 'b', 'c'})).as_dict
{'a': 'Bad', 'b': 'Excellent', 'c': 'Excellent'}
>>> converter('a > b > c > d').as_dict
{'a': 'Excellent', 'b': 'Very Good', 'c': 'Good', 'd': 'Bad'}
```

### 5.2.7 ConverterBallotToLevelsListNumeric

**class** whalrus.**ConverterBallotToLevelsListNumeric**(*scale: whalrus.scales.scale_from_list.ScaleFromList*, *borda_unordered_give_points: bool = True*)

Default converter to a *BallotLevels* using a *ScaleFromList* of numbers.

This converter works essentially the same as *ConverterBallotToLevelsInterval*, but it then maps the evaluations to levels of the scale.

**Parameters**

- **scale** (ScaleFromList) – The scale.
- **borda_unordered_give_points** (*bool*) – When converting a *BallotOrder* that is not a *BallotLevels*, we use Borda scores as a calculation step. This parameter decides whether the unordered candidates of the ballot give points to the ordered candidates. Cf. *ScorerBorda*.

#### Examples

Typical usages:

```
>>> converter = ConverterBallotToLevelsListNumeric(scale=ScaleFromList([-1, 0, 3,␣
↪4]))
>>> b = BallotLevels({'a': 1, 'b': .2}, candidates={'a', 'b', 'c'},␣
↪scale=ScaleInterval(-1, 1))
>>> converter(b).as_dict
{'a': 4, 'b': 3}
>>> b = BallotLevels({'a': 5, 'b': 4}, candidates={'a', 'b', 'c'},␣
↪scale=ScaleRange(0, 5))
>>> converter(b).as_dict
{'a': 4, 'b': 3}
>>> b = BallotLevels({'a': 4, 'b': 0}, candidates={'a', 'b', 'c'},␣
↪scale=ScaleFromSet({-1, 0, 4}))
>>> converter(b).as_dict
```

(continues on next page)

(continued from previous page)

```
{'a': 4, 'b': 0}
>>> converter(BallotOneName('a', candidates={'a', 'b', 'c'})).as_dict
{'a': 4, 'b': -1, 'c': -1}
>>> converter(BallotPlurality('a', candidates={'a', 'b', 'c'})).as_dict
{'a': 4, 'b': -1, 'c': -1}
>>> converter(BallotVeto('a', candidates={'a', 'b', 'c'})).as_dict
{'a': -1, 'b': 4, 'c': 4}
>>> converter('a > b > c > d').as_dict
{'a': 4, 'b': 3, 'c': 0, 'd': -1}
```

### 5.2.8 ConverterBallotToLevelsRange

**class** whalrus.**ConverterBallotToLevelsRange**(*scale: whalrus.scales.scale_range.ScaleRange = ScaleRange(low=0, high=1), borda_unordered_give_points: bool = True*)

Default converter to a *BallotLevels* using a *ScaleRange* (range of integers).

This converter works essentially the same as *ConverterBallotToLevelsInterval*, but it rounds the grades to the nearest integers.

**Parameters**

- **scale** (ScaleRange) – The scale.
- **borda_unordered_give_points** (*bool*) – When converting a *BallotOrder* that is not a *BallotLevels*, we use Borda scores (normalized to the interval [`scale.low`, `scale.high`] and rounded). This parameter decides whether the unordered candidates of the ballot give points to the ordered candidates. Cf. *ScorerBorda*.

#### Examples

Typical usages:

```
>>> converter = ConverterBallotToLevelsRange(scale=ScaleRange(low=0, high=10))
>>> b = BallotLevels({'a': 1, 'b': .4}, candidates={'a', 'b', 'c'},␣
↪scale=ScaleInterval(-1, 1))
>>> converter(b).as_dict
{'a': 10, 'b': 7}
>>> b = BallotLevels({'a': 5, 'b': 4}, candidates={'a', 'b', 'c'},␣
↪scale=ScaleRange(0, 5))
>>> converter(b).as_dict
{'a': 10, 'b': 8}
>>> b = BallotLevels({'a': 4, 'b': 0}, candidates={'a', 'b', 'c'},␣
↪scale=ScaleFromSet({-1, 0, 4}))
>>> converter(b).as_dict
{'a': 10, 'b': 2}
>>> b = BallotLevels(
...     {'a': 'Excellent', 'b': 'Very Good'}, candidates={'a', 'b', 'c'},
...     scale=ScaleFromList(['Bad', 'Medium', 'Good', 'Very Good', 'Great',
↪'Excellent']))
>>> converter(b).as_dict
{'a': 10, 'b': 6}
>>> converter(BallotOneName('a', candidates={'a', 'b', 'c'})).as_dict
{'a': 10, 'b': 0, 'c': 0}
```

(continues on next page)

(continued from previous page)

```
>>> converter(BallotPlurality('a', candidates={'a', 'b', 'c'})).as_dict
{'a': 10, 'b': 0, 'c': 0}
>>> converter(BallotVeto('a', candidates={'a', 'b', 'c'})).as_dict
{'a': 0, 'b': 10, 'c': 10}
>>> converter('a > b > c').as_dict
{'a': 10, 'b': 5, 'c': 0}
```

Options for converting ordered ballots:

```
>>> b = BallotOrder('a > b > c', candidates={'a', 'b', 'c', 'd', 'e', 'f'})
>>> converter = ConverterBallotToLevelsRange(scale=ScaleRange(low=0, high=10),
...                                          borda_unordered_give_points=False)
>>> converter(b).as_dict
{'a': 10, 'b': 5, 'c': 0}
>>> converter = ConverterBallotToLevelsRange(scale=ScaleRange(low=0, high=10),
...                                          borda_unordered_give_points=True)
>>> converter(b).as_dict
{'a': 10, 'b': 8, 'c': 6}
```

### 5.2.9 ConverterBallotToOrder

**class** whalrus.**ConverterBallotToOrder**

Default converter to a *BallotOrder*.

This is a default converter to a *BallotOrder*. It tries to infer the type of input and converts it to an ordered ballot (possibly a ballot of a subclass, such as *BallotLevels*).

#### Examples

```
>>> converter = ConverterBallotToOrder()
>>> converter('a > b ~ c')
BallotOrder(['a', {'b', 'c'}], candidates={'a', 'b', 'c'})
>>> converter(['a', {'b', 'c'}])
BallotOrder(['a', {'b', 'c'}], candidates={'a', 'b', 'c'})
>>> converter({'a': 10, 'b': 7, 'c': 0})
BallotLevels({'a': 10, 'b': 7, 'c': 0}, candidates={'a', 'b', 'c'}, scale=Scale())
>>> converter(BallotOneName('a', candidates={'a', 'b', 'c'}))
BallotOrder(['a', {'b', 'c'}], candidates={'a', 'b', 'c'})
>>> converter(BallotPlurality('a', candidates={'a', 'b', 'c'}))
BallotOrder(['a', {'b', 'c'}], candidates={'a', 'b', 'c'})
>>> converter(BallotVeto('a', candidates={'a', 'b', 'c'}))
BallotOrder([{'b', 'c'}, 'a'], candidates={'a', 'b', 'c'})
```

### 5.2.10 ConverterBallotToPlurality

**class** whalrus.**ConverterBallotToPlurality**(*priority: whalrus.priorities.priority.Priority = Priority.UNAMBIGUOUS, order_priority: whalrus.priorities.priority.Priority = None, plurality_priority: whalrus.priorities.priority.Priority = None, veto_priority: whalrus.priorities.priority.Priority = None, one_name_priority: whalrus.priorities.priority.Priority = None*)

Default converter to a *BallotPlurality*.

**Parameters**

- **priority** (Priority) – Serves as a default value for the other parameters if they are not explicitly mentioned. Default: *Priority.UNAMBIGUOUS*.
- **order_priority** (Priority) – Option passed to *BallotOrder.first()*. Default: priority.
- **plurality_priority** (Priority) – Option passed to *BallotPlurality.first()*. Default: priority.
- **veto_priority** (Priority) – Option passed to *BallotVeto.first()*. Default: priority.
- **one_name_priority** (Priority) – Option passed to *BallotOneName.first()*. Default: priority.

#### Examples

Typical usages:

```
>>> converter = ConverterBallotToPlurality()
>>> converter(BallotOneName('a', candidates={'a', 'b'}))
BallotPlurality('a', candidates={'a', 'b'})
>>> converter(BallotVeto('a', candidates={'a', 'b'}))
BallotPlurality('b', candidates={'a', 'b'})
>>> converter({'a': 10, 'b': 7, 'c':0})
BallotPlurality('a', candidates={'a', 'b', 'c'})
>>> converter('a > b ~ c')
BallotPlurality('a', candidates={'a', 'b', 'c'})
>>> converter(['a', {'b', 'c'}])
BallotPlurality('a', candidates={'a', 'b', 'c'})
```

Use options for the restrictions:

```
>>> converter = ConverterBallotToPlurality(priority=Priority.ASCENDING)
>>> converter('a ~ b > c')
BallotPlurality('a', candidates={'a', 'b', 'c'})
```

Misc:

```
>>> ballot = BallotVeto('a', candidates={'a', 'b', 'c'})
>>> converter = ConverterBallotToPlurality()
>>> converter(ballot, candidates={'a', 'b', 'd'})
BallotPlurality('b', candidates={'a', 'b'})
```

### 5.2.11 ConverterBallotToStrictOrder

**class** whalrus.**ConverterBallotToStrictOrder**(*priority: whalrus.priorities.priority.Priority = Priority.UNAMBIGUOUS*)

Default converter to a strictly ordered ballot.

This is a default converter to a strictly ordered ballot (cf. `BallotOrder.is_strict`). It tries to infer the type of input and converts it to a `BallotOrder` (possibly a ballot of a subclass, such as `BallotLevels`), ensuring that the represented order is strict.

**Parameters** **priority** (`Priority`) – The `Priority` used to break ties. Default: `Priority.UNAMBIGUOUS`.

#### Examples

```
>>> converter = ConverterBallotToStrictOrder(priority=Priority.ASCENDING)
>>> converter('a > b ~ c')
BallotOrder(['a', 'b', 'c'], candidates={'a', 'b', 'c'})
>>> converter(['a', {'b', 'c'}])
BallotOrder(['a', 'b', 'c'], candidates={'a', 'b', 'c'})
>>> converter({'a': 10, 'b': 7, 'c': 0})
BallotLevels({'a': 10, 'b': 7, 'c': 0}, candidates={'a', 'b', 'c'}, scale=Scale())
>>> converter(BallotOneName('a', candidates={'a', 'b', 'c'}))
BallotOrder(['a', 'b', 'c'], candidates={'a', 'b', 'c'})
>>> converter(BallotPlurality('a', candidates={'a', 'b', 'c'}))
BallotOrder(['a', 'b', 'c'], candidates={'a', 'b', 'c'})
>>> converter(BallotVeto('a', candidates={'a', 'b', 'c'}))
BallotOrder(['b', 'c', 'a'], candidates={'a', 'b', 'c'})
```

### 5.2.12 ConverterBallotToVeto

**class** whalrus.**ConverterBallotToVeto**(*priority: whalrus.priorities.priority.Priority = Priority.UNAMBIGUOUS, order_priority: whalrus.priorities.priority.Priority = None, plurality_priority: whalrus.priorities.priority.Priority = None, veto_priority: whalrus.priorities.priority.Priority = None, one_name_priority: whalrus.priorities.priority.Priority = None*)

Default converter to a `BallotVeto`.

**Parameters**

- **priority** (`Priority`) – Serves as a default value for the other parameters if they are not explicitly mentioned. Default: `Priority.UNAMBIGUOUS`.
- **order_priority** (`Priority`) – Option passed to `BallotOrder.last()`. Default: `priority`.
- **plurality_priority** (`Priority`) – Option passed to `BallotPlurality.last()`. Default: `priority`.
- **veto_priority** (`Priority`) – Option passed to `BallotVeto.last()`. Default: `priority`.
- **one_name_priority** (`Priority`) – Option passed to `BallotOneName.last()`. Default: `priority`.

### Examples

Typical usages:

```
>>> converter = ConverterBallotToVeto()
>>> converter(BallotOneName('a', candidates={'a', 'b'}))
BallotVeto('a', candidates={'a', 'b'})
>>> converter(BallotPlurality('a', candidates={'a', 'b'}))
BallotVeto('b', candidates={'a', 'b'})
>>> converter({'a': 10, 'b': 7, 'c':0})
BallotVeto('c', candidates={'a', 'b', 'c'})
>>> converter('a ~ b > c')
BallotVeto('c', candidates={'a', 'b', 'c'})
>>> converter([{'a', 'b'}, 'c'])
BallotVeto('c', candidates={'a', 'b', 'c'})
```

Use options for the restrictions:

```
>>> converter = ConverterBallotToVeto(priority=Priority.ASCENDING)
>>> converter('a > b ~ c')
BallotVeto('c', candidates={'a', 'b', 'c'})
```

# 5.3 Elimination

## 5.3.1 Elimination

`class whalrus.`**`Elimination`**`(*args, **kwargs)`

An elimination method.

An `Elimination` object is a callable whose input is a `Rule` (which has already loaded a profile). When the `Elimination` object is called, it loads the rule. The output of the call is the `Elimination` object itself. But after the call, you can access to the computed variables (ending with an underscore), such as `eliminated_order_`, `eliminated_` or `qualified_`.

**Parameters**

- **`args`** – If present, these parameters will be passed to `__call__` immediately after initialization.
- **`kwargs`** – If present, these parameters will be passed to `__call__` immediately after initialization.

**`rule_`**

This attribute stores the rule given in argument of the `__call__`.

**Type** *Rule*

### Examples

Cf. `EliminationLast` for some examples.

**`eliminated_`**

The eliminated candidates.

This should always be non-empty. It may contain all the candidates (for example, it is always the case when there was only one candidate in the election).

**Type** *NiceSet*

**eliminated_order_**

The order on the eliminated candidates.

It is a list where each element is a `NiceSet`. Each set represents a class of tied candidates. The first set in the list represents the "best" eliminated candidates, whereas the last set represent the "worst" candidates.

**Type** list

**qualified_**

The candidates that are qualified (not eliminated).

**Type** *NiceSet*

### 5.3.2 EliminationBelowAverage

**class** whalrus.**EliminationBelowAverage**(**args, strict=True, **kwargs*)

Elimination of the candidates whose score is lower than the average score

**Parameters**

- **args** – Cf. parent class.
- **strict** (*bool*) – If True (resp. False), then eliminate the candidates whose score is strictly lower than (resp. lower or equal to) the average.
- **kwargs** – Cf. parent class.

#### Examples

```
>>> rule = RulePlurality(ballots=['a', 'b', 'c', 'd'], weights=[35, 30, 25, 10])
>>> rule.gross_scores_
{'a': 35, 'b': 30, 'c': 25, 'd': 10}
>>> EliminationBelowAverage(rule=rule).eliminated_
{'d'}
>>> EliminationBelowAverage(rule=rule, strict=False).eliminated_
{'c', 'd'}
```

If no candidates should be eliminated (which may happen only if `strict` is True), then all candidates are eliminated.

```
>>> rule = RulePlurality(ballots=['a', 'b'])
>>> rule.gross_scores_
{'a': 1, 'b': 1}
>>> EliminationBelowAverage(rule=rule).eliminated_
{'a', 'b'}
```

**eliminated_**

The eliminated candidates.

This should always be non-empty. It may contain all the candidates (for example, it is always the case when there was only one candidate in the election).

**Type** *NiceSet*

**qualified_**

The candidates that are qualified (not eliminated).

**Type** *NiceSet*

### 5.3.3 EliminationLast

**class** whalrus.**EliminationLast**(**args, k: int = 1, **kwargs*)

Elimination of the last candidates (with a fixed number of candidates to eliminate, or to qualify).

**Parameters**

- **args** – Cf. parent class.
- **k** (*int*) – A nonzero integer. The number of eliminated candidates. If this number is negative, then `len(rule.candidates_) - abs(k)` candidates are eliminated, i.e. `abs(k)` candidates are qualified.
- **kwargs`** – Cf. parent class.

#### Examples

In the most general syntax, firstly, you define the elimination method:

```
>>> elimination = EliminationLast(k=1)
```

Secondly, you use it as a callable to load a particular election (rule, profile, candidates):

```
>>> rule = RulePlurality(ballots=['a', 'a', 'b', 'b', 'c'])
>>> elimination(rule)  # doctest:+ELLIPSIS
<... object at ...>
```

Finally, you can access the computed variables:

```
>>> elimination.eliminated_
{'c'}
```

Later, if you wish, you can load another election with the same elimination method, and so on.

Optionally, you can specify an election (rule, profile, candidates) as soon as the *Elimination* object is initialized. This allows for one-liners such as:

```
>>> EliminationLast(rule=RulePlurality(ballots=['a', 'a', 'b', 'b', 'c']), k=1).
↪eliminated_
{'c'}
```

Typical usage with `k = 1` (e.g. for *RuleIRV*):

```
>>> rule = RulePlurality(ballots=['a', 'a', 'a', 'b', 'b', 'c', 'c', 'd', 'e'],
...                      tie_break=Priority.ASCENDING)
>>> EliminationLast(rule=rule, k=1).eliminated_
{'e'}
```

Typical usage with `k = -2` (e.g. for *RuleTwoRound*):

```
>>> rule = RulePlurality(ballots=['a', 'a', 'a', 'b', 'b', 'c', 'c', 'd', 'e'],
...                      tie_break=Priority.ASCENDING)
>>> EliminationLast(rule=rule, k=-2).qualified_
{'a', 'b'}
```

Order of elimination:

```
>>> rule = RulePlurality(ballots=['a', 'a', 'a', 'b', 'b', 'c', 'c', 'd', 'e'],
...                      tie_break=Priority.ASCENDING)
>>> EliminationLast(rule=rule, k=-2).eliminated_order_
[{'c'}, {'d', 'e'}]
```

There must always be at least one eliminated candidate. If it is not possible to eliminate (case `k` > 0) or keep (case `k` < 0) as many candidates as required, then everybody is eliminated:

```
>>> rule = RulePlurality(ballots=['a'])
>>> EliminationLast(rule=rule, k=1).eliminated_
{'a'}
>>> EliminationLast(rule=rule, k=-2).eliminated_
{'a'}
```

**eliminated_**

The eliminated candidates.

This should always be non-empty. It may contain all the candidates (for example, it is always the case when there was only one candidate in the election).

**Type** *NiceSet*

**qualified_**

The candidates that are qualified (not eliminated).

**Type** *NiceSet*

## 5.4 Matrix

### 5.4.1 Matrix

**class** whalrus.**Matrix**(**args*, *converter: whalrus.converters_ballot.converter_ballot.ConverterBallot = None*, ***kwargs*)

A way to compute a matrix from a profile.

A *Matrix* object is a callable whose inputs are ballots and optionally weights, voters and candidates. When it is called, it loads the profile. The output of the call is the *Matrix* object itself. But after the call, you can access to the computed variables (ending with an underscore), such as *as_dict_* or *as_array_*.

**Parameters**

- **args** – If present, these parameters will be passed to `__call__` immediately after initialization.
- **converter** (`ConverterBallot`) – The converter that is used to convert input ballots in order to compute *profile_converted_*. Default: *ConverterBallotGeneral*.
- **kwargs** – If present, these parameters will be passed to `__call__` immediately after initialization.

**profile_original_**

The profile as it is entered by the user. This uses the constructor of *Profile*. Hence indirectly, it uses *ConverterBallotGeneral* to ensure, for example, that strings like `'a > b > c'` are converted to :class:`Ballot` objects.

**Type** *Profile*

**profile_converted_**

The profile, with ballots that are adequate for the voting rule. For example, in `MatrixWeightedMajority`, it will be `BallotOrder` objects. This uses the parameter `converter` of the object.

**Type** *Profile*

**candidates_**

The candidates of the election, as entered in the __`call`__.

**Type** *NiceSet*

### Examples

Cf. `MatrixWeightedMajority` for some examples.

**as_array_**

The matrix, as a numpy array. Each row and each column corresponds to a candidate (in the order of `candidates_as_list_`).

**Type** Array

**as_array_of_floats_**

The matrix, as a numpy array. It is the same as `as_array_`, but converted to floats.

**Type** Array

**as_dict_**

The matrix, as a `NiceDict`. Keys are pairs of candidates, and values are the coefficients of the matrix.

**Type** *NiceDict*

**candidates_as_list_**

The list of candidates. Candidates are sorted if possible.

**Type** list

**candidates_indexes_**

The candidates as a dictionary. To each candidate, it associates its index in `candidates_as_list_`.

**Type** *NiceDict*

## 5.4.2 MatrixMajority

**class** `whalrus.`**`MatrixMajority`**(**args, converter: whalrus.converters_ballot.converter_ballot.ConverterBallot = None, matrix_weighted_majority: whalrus.matrices.matrix.Matrix = None, greater: numbers.Number = 1, lower: numbers.Number = 0, equal: numbers.Number = Fraction(1, 2), diagonal: numbers.Number = Fraction(1, 2), **kwargs*)

The majority matrix.

**Parameters**

- **args** – Cf. parent class.
- **converter** (`ConverterBallot`) – Default: `ConverterBallotToOrder`.
- **matrix_weighted_majority** (`Matrix.`) – Algorithm used to compute the weighted majority matrix *W*. Default: `MatrixWeightedMajority`.
- **greater** (`Number`) – Value used when *W(c, d)* > *W(d, c)*.

- **lower** (*Number*) – Value used when *W(c, d)* < *W(d, c)*.
- **equal** (*Number*) – Value used when *W(c, d)* = *W(d, c)* (except for diagonal coefficients).
- **diagonal** (*Number*) – Value used for the diagonal coefficients.
- **kwargs** – Cf. parent class.

### Examples

First, we compute a matrix *W* with the algorithm given in the parameter `matrix_weighted_majority`. Then for each pair of candidates *(c, d)*, the coefficient of the majority matrix is set to `greater`, `lower`, `equal` or `diagonal`, depending on the values of *W(c, d)* and *W(d, c)*.

```
>>> MatrixMajority(ballots=['a > b ~ c', 'b > a > c', 'c > a > b']).as_array_
array([[Fraction(1, 2), 1, 1],
       [0, Fraction(1, 2), Fraction(1, 2)],
       [0, Fraction(1, 2), Fraction(1, 2)]], dtype=object)
```

Using the options:

```
>>> MatrixMajority(ballots=['a > b ~ c', 'b > a > c', 'c > a > b'], equal=0,␣
↪diagonal=0).as_array_
array([[0, 1, 1],
       [0, 0, 0],
       [0, 0, 0]])
```

**as_array_**

The matrix, as a numpy array. Each row and each column corresponds to a candidate (in the order of `candidates_as_list_`).

**Type** Array

**as_array_of_floats_**

The matrix, as a numpy array. It is the same as *as_array_*, but converted to floats.

**Type** Array

**matrix_weighted_majority_**

The weighted majority matrix (upon which the computation of the majority matrix is based), once computed with the given profile.

**Type** *Matrix*

## 5.4.3 MatrixRankedPairs

**class** whalrus.**MatrixRankedPairs**(**args*, *converter: whalrus.converters_ballot.converter_ballot.ConverterBallot = None*, *matrix_weighted_majority: whalrus.matrices.matrix.Matrix = None*, *tie_break: whalrus.priorities.priority.Priority = Priority.UNAMBIGUOUS*, ***kwargs*)

The ranked pairs matrix.

**Parameters**

- **args** – Cf. parent class.
- **converter** (ConverterBallot) – Default: *ConverterBallotToOrder*.

- **`matrix_weighted_majority`** (`Matrix`) – Algorithm used to compute the weighted majority matrix *W*. Default: *`MatrixWeightedMajority`*.
- **`tie_break`** (`Priority`) – The tie-break used when two duels have the same score.
- **`kwargs`** – Cf. parent class.

### Examples

First, we compute a matrix *W* with the algorithm given in the parameter `matrix_weighted_majority`. The ranked pair matrix represents a graph whose vertices are the candidates. In order to build it, we consider all duels between two distinct candidates *(c, d)*, by decreasing order of the value *W(c, d)*. We add an edge *(c, d)* in the ranked pairs matrix, except if it creates a cycle in the graph, and we consider the transitive closure.

```
>>> m = MatrixRankedPairs(['a > b > c', 'b > c > a', 'c > a > b'], weights=[4, 3,␣
↪2])
>>> m.edges_order_
[('b', 'c'), ('a', 'b'), ('c', 'a')]
>>> m.as_array_
array([[0, 1, 1],
       [0, 0, 1],
       [0, 0, 0]], dtype=object)
```

In the example example above, the edge *(b, c)* is added. Then it is the edge *(a, b)* which, by transitive closure, also adds the edge *(a, c)*. Finally the edge *(c, a)* (representing the victory of *c* over *a* in the weighted majority matrix) should be added, but it would introduce a cycle in the graph, so it is ignored.

If two duels have the same score, the tie-break is used. For example, with *`Priority.ASCENDING`*, we add a victory *(a, . . . )* before a victory *(b, . . . )*; and we add a victory *(a, c)* before a victory *(a, b)* (because *b* is favored over *c*). A very simple but illustrative example:

```
>>> MatrixRankedPairs(['a > b > c'], tie_break=Priority.ASCENDING).edges_order_
[('a', 'c'), ('a', 'b'), ('b', 'c')]
```

**`as_array_of_floats_`**

The matrix, as a numpy array. It is the same as `as_array_`, but converted to floats.

**Type** Array

**`edges_order_`**

The order in which edges should be added (if possible). It is a list of pairs of candidates. E.g. `[('b', 'c'), ('c', 'a'), ('a', 'b')]`, where ('b', 'c') is the first edge to add.

**Type** list

**`matrix_weighted_majority_`**

The weighted majority matrix (upon which the computation of the Ranked Pairs matrix is based), once computed with the given profile).

**Type** *Matrix*

## 5.4.4 MatrixSchulze

**class** `whalrus.`**`MatrixSchulze`**(**args*, *converter: whalrus.converters_ballot.converter_ballot.ConverterBallot = None*, *matrix_weighted_majority: whalrus.matrices.matrix.Matrix = None*, ***kwargs*)

The Schulze matrix.

**Parameters**

- **args** – Cf. parent class.
- **converter** (ConverterBallot) – Default: *ConverterBallotToOrder*.
- **matrix_weighted_majority** (Matrix) – Algorithm used to compute the weighted majority matrix *W*. Default: *MatrixWeightedMajority*.
- **kwargs** – Cf. parent class.

### Examples

First, we compute a matrix *W* with the algorithm given in the parameter `matrix_weighted_majority`. The Schulze matrix gives, for each pair of candidates *(c, d)*, the width of the widest path from *c* to *d*, where the width of a path is the minimum weight of its edges.

```
>>> m = MatrixSchulze(['a > b > c', 'b > c > a', 'c > a > b'], weights=[4, 3, 2])
>>> m.as_array_
array([[0, Fraction(2, 3), Fraction(2, 3)],
       [Fraction(5, 9), 0, Fraction(7, 9)],
       [Fraction(5, 9), Fraction(5, 9), 0]], dtype=object)
```

**as_array_of_floats_**

The matrix, as a numpy array. It is the same as `as_array_`, but converted to floats.

**Type** Array

**matrix_weighted_majority_**

The weighted majority matrix (upon which the computation of the Schulze is based), once computed with the given profile.

**Type** *Matrix*

## 5.4.5 MatrixWeightedMajority

**class** whalrus.**MatrixWeightedMajority**(**args, converter: whalrus.converters_ballot.converter_ballot.ConverterBallot = None, higher_vs_lower: Optional[numbers.Number] = 1, lower_vs_higher: Optional[numbers.Number] = 0, indifference: Optional[numbers.Number] = Fraction(1, 2), ordered_vs_unordered: Optional[numbers.Number] = 1, unordered_vs_ordered: Optional[numbers.Number] = 0, unordered_vs_unordered: Optional[numbers.Number] = Fraction(1, 2), ordered_vs_absent: Optional[numbers.Number] = None, absent_vs_ordered: Optional[numbers.Number] = None, unordered_vs_absent: Optional[numbers.Number] = None, absent_vs_unordered: Optional[numbers.Number] = None, absent_vs_absent: Optional[numbers.Number] = None, diagonal_score: numbers.Number = 0, default_score: numbers.Number = 0, antisymmetric: bool = False, **kwargs*)

The weighted majority matrix.

**Parameters**

- **args** – Cf. parent class.
- **converter** (ConverterBallot) – Default: *ConverterBallotToOrder*.
- **higher_vs_lower** (*Number or None*) – Number of points for candidate *c* when it is ordered higher than candidate *d*.
- **lower_vs_higher** (*Number or None*) – Number of points for candidate *c* when it is ordered lower than candidate *d*.
- **indifference** (*Number or None*) – Number of points for candidate *c* when it is ordered and tied with candidate *d*.
- **ordered_vs_unordered** (*Number or None*) – Number of points for candidate *c* when it is ordered and *d* is unordered.
- **unordered_vs_ordered** (*Number or None*) – Number of points for candidate *c* when it is unordered and *d* is ordered.
- **unordered_vs_unordered** (*Number or None*) – Number of points for candidate *c* when it is unordered and *d* is unordered.
- **ordered_vs_absent** (*Number or None*) – Number of points for candidate *c* when it is ordered and *d* is absent.
- **absent_vs_ordered** (*Number or None*) – Number of points for candidate *c* when it is absent and *d* is ordered.
- **unordered_vs_absent** (*Number or None*) – Number of points for candidate *c* when it is unordered and *d* is absent.
- **absent_vs_unordered** (*Number or None*) – Number of points for candidate *c* when it is absent and *d* is unordered.
- **absent_vs_absent** (*Number or None*) – Number of points for candidate *c* when it is absent and *d* is absent.
- **diagonal_score** (*Number*) – Value of the diagonal coefficients.
- **default_score** (*Number*) – Default score in the matrix in case of division by 0 (except for the diagonal coefficients).
- **antisymmetric** (*bool*) – If True, then an antisymmetric version of the matrix is computed (by subtracting the transposed matrix at the end of the computation).
- **kwargs** – Cf. parent class.

## Examples

In the most general syntax, firstly, you define the matrix computation algorithm:

```
>>> matrix = MatrixWeightedMajority(diagonal_score=.5)
```

Secondly, you use it as a callable to load a particular election (profile, candidates):

```
>>> matrix(ballots=['a > b', 'b > a'], weights=[3, 1], voters=['v', 'w'],␣
↪candidates={'a', 'b'})  # doctest:+ELLIPSIS
<... object at ...>
```

Finally, you can access the computed variables:

```
>>> matrix.as_array_
array([[Fraction(1, 2), Fraction(3, 4)],
       [Fraction(1, 4), Fraction(1, 2)]], dtype=object)
```

Later, if you wish, you can load another profile with the same matrix computation algorithm, and so on.

Optionally, you can specify an election (profile and candidates) as soon as the `Matrix` object is initialized. This allows for "one-liners" such as:

```
>>> MatrixWeightedMajority(ballots=['a > b', 'b > a'], weights=[3, 1], voters=['x
↪', 'y'],
...                        candidates={'a', 'b'}, diagonal_score=.5).as_array_
array([[Fraction(1, 2), Fraction(3, 4)],
       [Fraction(1, 4), Fraction(1, 2)]], dtype=object)
```

Antisymmetric version:

```
>>> MatrixWeightedMajority(ballots=['a > b', 'b > a'], weights=[3, 1], voters=['x
↪', 'y'],
...                        candidates={'a', 'b'}, antisymmetric=True).as_array_
array([[0, Fraction(1, 2)],
       [Fraction(-1, 2), 0]], dtype=object)
```

An "unordered" candidate is a candidate that the voter has seen but not included in her ranking; i.e. it is in the attribute `BallotOrder.candidates_not_in_b` of the ballot. An "absent" candidate is a candidate that the voter has not even seen; i.e. it is in `self.candidates_`, but not the attribute `Ballot.candidates` of the ballot. For all the "scoring" parameters (from `higher_vs_lower` to `absent_vs_absent`), the value None can be used. In that case, the corresponding occurrences are not taken into account in the average (neither the numerator, not the denominator). Consider this example:

```
>>> ballots = ['a > b', 'a ~ b']
```

With `indifference=Fraction(1, 2)` (default), the ratio of voters who prefer *a* to *b* is (1 + 1 / 2) / 2 = 3 / 4 (the indifferent voter gives 1 / 2 point and is counted in the denominator):

```
>>> MatrixWeightedMajority(ballots).as_array_
array([[0, Fraction(3, 4)],
       [Fraction(1, 4), 0]], dtype=object)
```

With `indifference=0`, the ratio of voters who prefer *a* to *b* is 1 / 2 (the indifferent voter gives no point, but is counted in the denominator):

```
>>> MatrixWeightedMajority(ballots, indifference=0).as_array_
array([[0, Fraction(1, 2)],
       [0, 0]], dtype=object)
```

With `indifference=None`, the ratio of voters who prefer *a* to *b* is 1 / 1 = 1 (the indifferent voter is not counted in the average at all):

```
>>> MatrixWeightedMajority(ballots, indifference=None).as_array_
array([[0, 1],
       [0, 0]])
```

**`as_array_`**

The matrix, as a numpy array. Each row and each column corresponds to a candidate (in the order of `candidates_as_list_`).

**Type** Array

**as_array_of_floats_**

The matrix, as a numpy array. It is the same as *as_array_*, but converted to floats.

**Type** Array

**candidates_as_list_**

The list of candidates. Candidates are sorted if possible.

**Type** list

**candidates_indexes_**

The candidates as a dictionary. To each candidate, it associates its index in *candidates_as_list_*.

**Type** *NiceDict*

**gross_**

The "gross" matrix. Keys are pairs of candidates. Each coefficient is the weighted number of points (used as numerator in the average).

### Examples

```
>>> from whalrus import MatrixWeightedMajority
>>> MatrixWeightedMajority(ballots=['a > b', 'a ~ b'], weights=[2, 1]).gross_
{('a', 'a'): 0, ('a', 'b'): Fraction(5, 2), ('b', 'a'): Fraction(1, 2), ('b',
↪'b'): 0}
```

**Type** *NiceDict*

**weights_**

The matrix of weights. Keys are pairs of candidates. Each coefficient is the total weight (used as denominator in the average).

### Examples

In most usual cases, all non-diagonal coefficients are equal, and are equal to the total weight of all voters:

```
>>> from whalrus import MatrixWeightedMajority
>>> MatrixWeightedMajority(ballots=['a > b', 'a ~ b'], weights=[2, 1]).
↪weights_
{('a', 'a'): 0, ('a', 'b'): 3, ('b', 'a'): 3, ('b', 'b'): 0}
```

However, if some scoring parameters are None, some weights can be lower than the total weight of all voters:

```
>>> from whalrus import MatrixWeightedMajority
>>> MatrixWeightedMajority(ballots=['a > b', 'a ~ b'], weights=[2, 1],
...                        indifference=None).weights_
{('a', 'a'): 0, ('a', 'b'): 2, ('b', 'a'): 2, ('b', 'b'): 0}
```

**Type** *NiceDict*

## 5.5 Priority

### 5.5.1 Priority

**class** whalrus.**Priority**(*name: str*)

A priority setting, i.e. a policy to break ties and indifference classes.

**Parameters** **name** (*str*) – The name of this priority setting.

**UNAMBIGUOUS**

Shortcut for *PriorityUnambiguous*.

**ABSTAIN**

Shortcut for *PriorityAbstain*.

**ASCENDING**

Shortcut for *PriorityAscending*.

**DESCENDING**

Shortcut for *PriorityDescending*.

**RANDOM**

Shortcut for *PriorityRandom*.

#### Examples

Typical usage:

```
>>> priority = Priority.ASCENDING
>>> priority.choice({'c', 'a', 'b'})
'a'
>>> priority.sort({'c', 'a', 'b'})
['a', 'b', 'c']
```

**choice**(*x: Union[set, list], reverse: bool = False*) → object

Choose an element from a list, set, etc.

**Parameters**

- **x** (*list, set, etc.*) – The list, set, etc where the element is to be chosen.
- **reverse** (*bool*) – If False (default), then we choose the “first” or “best” element in this priority order. For example, if this is the ascending priority, we choose the lowest element. If True, then we choose the “last” or “worst” element. This is used, for example, in *RuleVeto*.

**Returns** The chosen element (or None). When x is empty, return None. When x has one element, return this element.

**Return type** object

**compare**(*c, d*) → int

Compare two candidates.

**Parameters**

- **c** (*candidate*) –
- **d** (*candidate.*) –

**Returns** 0 if *c* = *d*, -1 if the tie is broken in favor of *c* over *d*, 1 otherwise.

**Return type** int

**sort** (*x: Union[set, list], reverse: bool = False*) → Optional[list]

Sort a list, set, etc.

The original list `x` is not modified.

**Parameters**

- **x** (*list, set, etc.*) –
- **reverse** (*bool*) – If True, we use the reverse priority order.

**Returns** A sorted list (or None).

**Return type** list or None

**sort_pairs_rp** (*x: Union[set, list], reverse: bool = False*) → Optional[list]

Sort a list, set, etc. of pairs of candidates (for Ranked Pairs).

By default, it is in the normal priority order for the first element of the pair, and in the reverse priority order for the second element of the pair.

The original list `x` is not modified.

**Parameters**

- **x** (*list, set, etc.*) –
- **reverse** (*bool*) – If True, we use the reverse priority order.

**Returns** A sorted list (or None).

**Return type** list or None

## 5.5.2 PriorityAbstain

**class** whalrus.**PriorityAbstain**

When there are two elements or more, return None.

### Examples

```
>>> print(Priority.ABSTAIN.choice({'a', 'b'}))
None
>>> print(Priority.ABSTAIN.sort({'a', 'b'}))
None
```

**choice** (*x: Union[set, list], reverse: bool = False*) → object

Choose an element from a list, set, etc.

**Parameters**

- **x** (*list, set, etc.*) – The list, set, etc where the element is to be chosen.
- **reverse** (*bool*) – If False (default), then we choose the “first” or “best” element in this priority order. For example, if this is the ascending priority, we choose the lowest element. If True, then we choose the “last” or “worst” element. This is used, for example, in *RuleVeto*.

**Returns** The chosen element (or None). When `x` is empty, return None. When `x` has one element, return this element.

**Return type** object

**compare** (*c*, *d*) → int

Compare two candidates.

**Parameters**

- **c** (*candidate*) –
- **d** (*candidate.*) –

**Returns** 0 if *c* = *d*, -1 if the tie is broken in favor of *c* over *d*, 1 otherwise.

**Return type** int

**sort** (*x: Union[set, list], reverse: bool = False*) → Optional[list]

Sort a list, set, etc.

The original list x is not modified.

**Parameters**

- **x** (*list, set, etc.*) –
- **reverse** (*bool*) – If True, we use the reverse priority order.

**Returns** A sorted list (or None).

**Return type** list or None

**sort_pairs_rp** (*x: Union[set, list], reverse: bool = False*) → Optional[list]

Sort a list, set, etc. of pairs of candidates (for Ranked Pairs).

By default, it is in the normal priority order for the first element of the pair, and in the reverse priority order for the second element of the pair.

The original list x is not modified.

**Parameters**

- **x** (*list, set, etc.*) –
- **reverse** (*bool*) – If True, we use the reverse priority order.

**Returns** A sorted list (or None).

**Return type** list or None

### 5.5.3 PriorityAscending

**class** whalrus.**PriorityAscending**

Ascending order (lowest is favoured).

#### Examples

```
>>> Priority.ASCENDING.choice({'a', 'b'})
'a'
>>> Priority.ASCENDING.sort({'a', 'b'})
['a', 'b']
>>> Priority.ASCENDING.sort_pairs_rp({('a', 'b'), ('b', 'a'), ('a', 'c')})
[('a', 'c'), ('a', 'b'), ('b', 'a')]
```

**choice** (*x: Union[set, list], reverse: bool = False*) → object

Choose an element from a list, set, etc.

**Parameters**

- **x** (*list, set, etc.*) – The list, set, etc where the element is to be chosen.
- **reverse** (*bool*) – If False (default), then we choose the "first" or "best" element in this priority order. For example, if this is the ascending priority, we choose the lowest element. If True, then we choose the "last" or "worst" element. This is used, for example, in *RuleVeto*.

**Returns** The chosen element (or None). When `x` is empty, return None. When `x` has one element, return this element.

**Return type** object

**compare** (*c*, *d*) → int

Compare two candidates.

**Parameters**

- **c** (*candidate*) –
- **d** (*candidate.*) –

**Returns** 0 if $c = d$, -1 if the tie is broken in favor of $c$ over $d$, 1 otherwise.

**Return type** int

**sort** (*x: Union[set, list], reverse: bool = False*) → Optional[list]

Sort a list, set, etc.

The original list `x` is not modified.

**Parameters**

- **x** (*list, set, etc.*) –
- **reverse** (*bool*) – If True, we use the reverse priority order.

**Returns** A sorted list (or None).

**Return type** list or None

**sort_pairs_rp** (*x: Union[set, list], reverse: bool = False*) → Optional[list]

Sort a list, set, etc. of pairs of candidates (for Ranked Pairs).

By default, it is in the normal priority order for the first element of the pair, and in the reverse priority order for the second element of the pair.

The original list `x` is not modified.

**Parameters**

- **x** (*list, set, etc.*) –
- **reverse** (*bool*) – If True, we use the reverse priority order.

**Returns** A sorted list (or None).

**Return type** list or None

### 5.5.4 PriorityDescending

**class** whalrus.**PriorityDescending**

Descending order (highest is favoured).

#### Examples

```
>>> Priority.DESCENDING.choice({'a', 'b'})
'b'
>>> Priority.DESCENDING.sort({'a', 'b'})
['b', 'a']
>>> Priority.DESCENDING.sort_pairs_rp({('a', 'b'), ('b', 'a'), ('a', 'c')})
[('b', 'a'), ('a', 'b'), ('a', 'c')]
```

**choice** (*x: Union[set, list], reverse: bool = False*) → object

Choose an element from a list, set, etc.

**Parameters**

- **x** (*list, set, etc.*) – The list, set, etc where the element is to be chosen.
- **reverse** (*bool*) – If False (default), then we choose the "first" or "best" element in this priority order. For example, if this is the ascending priority, we choose the lowest element. If True, then we choose the "last" or "worst" element. This is used, for example, in *RuleVeto*.

**Returns** The chosen element (or None). When x is empty, return None. When x has one element, return this element.

**Return type** object

**compare** (*c*, *d*) → int

Compare two candidates.

**Parameters**

- **c** (*candidate*) –
- **d** (*candidate.*) –

**Returns** 0 if *c* = *d*, -1 if the tie is broken in favor of *c* over *d*, 1 otherwise.

**Return type** int

**sort** (*x: Union[set, list], reverse: bool = False*) → Optional[list]

Sort a list, set, etc.

The original list x is not modified.

**Parameters**

- **x** (*list, set, etc.*) –
- **reverse** (*bool*) – If True, we use the reverse priority order.

**Returns** A sorted list (or None).

**Return type** list or None

**sort_pairs_rp** (*x: Union[set, list], reverse: bool = False*) → Optional[list]

Sort a list, set, etc. of pairs of candidates (for Ranked Pairs).

By default, it is in the normal priority order for the first element of the pair, and in the reverse priority order for the second element of the pair.

The original list x is not modified.

**Parameters**

- **x** (*list, set, etc.*) –
- **reverse** (*bool*) – If True, we use the reverse priority order.

**Returns** A sorted list (or None).

**Return type** list or None

### 5.5.5 PriorityRandom

**class** whalrus.**PriorityRandom**

Random order.

#### Examples

```
>>> my_choice = Priority.RANDOM.choice({'a', 'b'})
>>> my_choice in {'a', 'b'}
True
>>> my_order = Priority.RANDOM.sort({'a', 'b'})
>>> my_order == ['a', 'b'] or my_order == ['b', 'a']
True
```

**choice** (*x: Union[set, list], reverse: bool = False*) → object

Choose an element from a list, set, etc.

**Parameters**

- **x** (*list, set, etc.*) – The list, set, etc where the element is to be chosen.
- **reverse** (*bool*) – If False (default), then we choose the "first" or "best" element in this priority order. For example, if this is the ascending priority, we choose the lowest element. If True, then we choose the "last" or "worst" element. This is used, for example, in *RuleVeto*.

**Returns** The chosen element (or None). When x is empty, return None. When x has one element, return this element.

**Return type** object

**compare** (*c, d*) → int

Compare two candidates.

**Parameters**

- **c** (*candidate*) –
- **d** (*candidate.*) –

**Returns** 0 if *c* = *d*, -1 if the tie is broken in favor of *c* over *d*, 1 otherwise.

**Return type** int

**sort** (*x: Union[set, list], reverse: bool = False*) → Optional[list]

Sort a list, set, etc.

The original list `x` is not modified.

**Parameters**

- **x** (*list, set, etc.*) –
- **reverse** (*bool*) – If True, we use the reverse priority order.

**Returns** A sorted list (or None).

**Return type** list or None

**sort_pairs_rp** (*x: Union[set, list], reverse: bool = False*) → Optional[list]

Sort a list, set, etc. of pairs of candidates (for Ranked Pairs).

By default, it is in the normal priority order for the first element of the pair, and in the reverse priority order for the second element of the pair.

The original list `x` is not modified.

**Parameters**

- **x** (*list, set, etc.*) –
- **reverse** (*bool*) – If True, we use the reverse priority order.

**Returns** A sorted list (or None).

**Return type** list or None

### 5.5.6 PriorityUnambiguous

**class** whalrus.**PriorityUnambiguous**

When there are two elements or more, raise a ValueError.

#### Examples

```
>>> try:
...     Priority.UNAMBIGUOUS.choice({'a', 'b'})
... except ValueError:
...     print('Cannot choose')
Cannot choose
>>> try:
...     Priority.UNAMBIGUOUS.sort({'a', 'b'})
... except ValueError:
...     print('Cannot sort')
Cannot sort
```

**choice** (*x: Union[set, list], reverse: bool = False*) → object

Choose an element from a list, set, etc.

**Parameters**

- **x** (*list, set, etc.*) – The list, set, etc where the element is to be chosen.
- **reverse** (*bool*) – If False (default), then we choose the “first” or “best” element in this priority order. For example, if this is the ascending priority, we choose the lowest element. If True, then we choose the “last” or “worst” element. This is used, for example, in *RuleVeto*.

**Returns** The chosen element (or None). When `x` is empty, return None. When `x` has one element, return this element.

**Return type** object

**compare** (*c*, *d*) → int

Compare two candidates.

**Parameters**

- **c** (*candidate*) –
- **d** (*candidate.*) –

**Returns** 0 if *c* = *d*, -1 if the tie is broken in favor of *c* over *d*, 1 otherwise.

**Return type** int

**sort** (*x: Union[set, list], reverse: bool = False*) → Optional[list]

Sort a list, set, etc.

The original list `x` is not modified.

**Parameters**

- **x** (*list, set, etc.*) –
- **reverse** (*bool*) – If True, we use the reverse priority order.

**Returns** A sorted list (or None).

**Return type** list or None

**sort_pairs_rp** (*x: Union[set, list], reverse: bool = False*) → Optional[list]

Sort a list, set, etc. of pairs of candidates (for Ranked Pairs).

By default, it is in the normal priority order for the first element of the pair, and in the reverse priority order for the second element of the pair.

The original list `x` is not modified.

**Parameters**

- **x** (*list, set, etc.*) –
- **reverse** (*bool*) – If True, we use the reverse priority order.

**Returns** A sorted list (or None).

**Return type** list or None

## 5.6 Profile

**class** whalrus.**Profile** (*ballots: Union[list, Profile], weights: list = None, voters: list = None*)

A profile of ballots.

**Parameters**

- **ballots** (*iterable*) – Typically, it is a list, but it can also be a `Profile`. Its elements must be `Ballot` objects or, more generally, inputs that can be interpreted by `ConverterBallotGeneral`.
- **weights** (*list*) – A list of numbers representing the weights of the ballots. Default: if `ballots` is a Profile, then use the weights of this profile; otherwise, all weights are 1.

- **voters** (*list*) – A list representing the voters corresponding to the ballots. Default: if `ballots` is a Profile, then use the voters of this profile; otherwise, all voters are None.

### Examples

Most general syntax:

```
>>> profile = Profile(
...     ballots=[BallotOrder('a > b ~ c'), BallotOrder('a ~ b > c')],
...     weights=[2, 1],
...     voters=['Alice', 'Bob']
... )
>>> print(profile)
Alice (2): a > b ~ c
Bob (1): a ~ b > c
```

In the following example, each ballot illustrates a different syntax:

```
>>> profile = Profile([
...     ['a', 'b', 'c'],
...     ('b', 'c', 'a'),
...     'c > a > b',
... ])
>>> print(profile)
a > b > c
b > c > a
c > a > b
```

Profiles have a list-like behavior in the sense that they implement `__len__`, `__getitem__`, `__setitem__` and `__delitem__`:

```
>>> profile = Profile(['a > b', 'b > a', 'a ~ b'])
>>> len(profile)
3
>>> profile[0]
BallotOrder(['a', 'b'], candidates={'a', 'b'})
>>> profile[0] = 'a ~ b'
>>> print(profile)
a ~ b
b > a
a ~ b
>>> del profile[0]
>>> print(profile)
b > a
a ~ b
```

Profiles can be concatenated:

```
>>> profile = Profile(['a > b', 'b > a']) + ['a ~ b']
>>> print(profile)
a > b
b > a
a ~ b
```

Profiles can be multiplied by a scalar, which multiplies the weights:

```
>>> profile = Profile(['a > b', 'b > a']) * 3
>>> print(profile)
(3): a > b
(3): b > a
```

**append**(*ballot: object*, *weight: numbers.Number = 1*, *voter: object = None*) → None

Append a ballot to the profile.

**Parameters**

- **ballot** (*object*) – A ballot or, more generally, an input that can be interpreted by *ConverterBallotGeneral*.
- **weight** (*Number*) – The weight of the ballot.
- **voter** (*object*) – The voter.

### Examples

```
>>> profile = Profile(['a > b'])
>>> profile.append('b > a')
>>> print(profile)
a > b
b > a
```

**ballots**

The ballots.

### Examples

```
>>> profile = Profile(['a > b', 'b > a'])
>>> profile.ballots
[BallotOrder(['a', 'b'], candidates={'a', 'b'}), BallotOrder(['b', 'a'],␣
↪candidates={'a', 'b'})]
```

**Type** list of Ballot

**has_voters**

Presence of explicit voters. True iff at least one voter is not None.

### Examples

```
>>> profile = Profile(['a > b', 'b > a'])
>>> profile.has_voters
False
```

**Type** bool

**has_weights**

Presence of non-trivial weights. True iff at least one weight is not 1.

### Examples

```
>>> profile = Profile(['a > b', 'b > a'])
>>> profile.has_weights
False
```

**Type** bool

**items**() → Iterator[T_co]

Items of the profile.

**Returns** A zip of triples (ballot, weight, voter).

**Return type** Iterator

### Examples

```
>>> profile = Profile(['a > b', 'b > a'])
>>> for ballot, weight, voter in profile.items():
...     print('Ballot %s, weight %s, voter %s.' % (ballot, weight, voter))
Ballot a > b, weight 1, voter None.
Ballot b > a, weight 1, voter None.
```

**remove**(*ballot: object = None*, *voter: object = None*) → None

Remove a ballot from the profile.

If only the ballot is specified, remove the first matching ballot in the profile. If only the voter is specified, remove the first ballot whose voter matches the given voter. If both are specified, remove the first ballot matching both descriptions.

**Parameters**

- **ballot** (*object*) – The ballot or, more generally, an input that can be interpreted by *ConverterBallotGeneral*.
- **voter** (*object*) – The voter.

### Examples

```
>>> profile = Profile(['a > b', 'b > a'])
>>> profile.remove('b > a')
>>> print(profile)
a > b
```

**voters**

The voters.

### Examples

```
>>> profile = Profile(['a > b', 'b > a'], voters=['Alice', 'Bob'])
>>> profile.voters
['Alice', 'Bob']
```

**Type** list

**weights**

The weights.

### Examples

```
>>> profile = Profile(['a > b', 'b > a'])
>>> profile.weights
[1, 1]
```

**Type** list of Number

## 5.7 Rule: In General

### 5.7.1 Rule

**class** whalrus.**Rule** (**args, tie_break: whalrus.priorities.priority.Priority = Priority.UNAMBIGUOUS, converter: whalrus.converters_ballot.converter_ballot.ConverterBallot = None, **kwargs*)

A voting rule.

A *Rule* object is a callable whose inputs are ballots and optionally weights, voters and candidates. When the rule is called, it loads the profile. The output of the call is the rule itself. But after the call, you can access to the computed variables (ending with an underscore), such as *cowinners_*.

At the initialization of a *Rule* object, some options can be given, such as a tie-break rule or a converter. In some subclasses, there can also be an option about the way to count abstentions, etc.

**Parameters**

- **args** – If present, these parameters will be passed to `__call__` immediately after initialization.
- **tie_break** (`Priority`) – A tie-break rule.
- **converter** (`ConverterBallot`) – The converter that is used to convert input ballots in order to compute *profile_converted_*. Default: *ConverterBallotGeneral*.
- **kwargs** – If present, these parameters will be passed to `__call__` immediately after initialization.

**profile_original_**

The profile as it is entered by the user. Since it uses the constructor of *Profile*, it indirectly uses *ConverterBallotGeneral* to ensure, for example, that strings like `'a > b > c'` are converted to *Ballot* objects.

**Type** *Profile*

**profile_converted_**

The profile, with ballots that are adapted to the voting rule. For example, in *RulePlurality*, it will be *BallotPlurality* objects, even if the original ballots are *BallotOrder* objects. This uses the parameter `converter` of the rule.

**Type** *Profile*

**candidates_**

The candidates of the election, as entered in the `__call__`.

**Type** *NiceSet*

### Examples

Cf. `RulePlurality` for some examples.

**`cotrailers_`**

"Cotrailers" of the election, i.e. the candidates that fare worst in the election. This is the last equivalence class in `order_`. For example, in `RuleScoreNum`, it is the candidates that are tied for the worst score.

**Type** *NiceSet*

**`cowinners_`**

Cowinners of the election, i.e. the candidates that fare best in the election.. This is the first equivalence class in `order_`. For example, in `RuleScoreNum`, it is the candidates that are tied for the best score.

**Type** *NiceSet*

**`n_candidates_`**

Number of candidates.

**Type** int

**`order_`**

Result of the election as a (weak) order over the candidates. This is a list of `NiceSet`. The first set contains the candidates that are tied for victory, etc.

**Type** list

**`strict_order_`**

Result of the election as a strict order over the candidates. The first element is the winner, etc. This may use the tie-breaking rule.

**Type** list

**`trailer_`**

The "trailer" of the election. This is the last candidate in `strict_order_` and also the unfavorable choice of the tie-breaking rule in `cotrailers_`.

**Type** object

**`winner_`**

The winner of the election. This is the first candidate in `strict_order_` and also the choice of the tie-breaking rule in `cowinners_`.

**Type** object

## 5.7.2 RuleIteratedElimination

**class** `whalrus.`**`RuleIteratedElimination`**(**args, base_rule: whalrus.rules.rule.Rule = None, elimination: whalrus.eliminations.elimination.Elimination = None, propagate_tie_break=True, **kwargs*)

A rule by iterated elimination (such as `RuleIRV`, `RuleCoombs`, `RuleNanson`, etc.)

**Parameters**

- **`args`** – Cf. parent class.

- **base_rule** (Rule) – The rule used at each round to determine the eliminated candidate(s). Unlike for *RuleSequentialElimination*, all the rounds use the same voting rule.
- **elimination** (Elimination) – The elimination algorithm. Default: `EliminationLast(k=1)`.
- **propagate_tie_break** (*bool*) – If True (default), then the tie-breaking rule of this object is also used for the base rule (cf. below).
- **kwargs** – Cf. parent class.

### Examples

```
>>> irv = RuleIteratedElimination(['a > b > c', 'b > a > c', 'c > a > b'],␣
↪weights=[2, 3, 4],
...                                base_rule=RulePlurality())
>>> irv.eliminations_[0].rule_.gross_scores_
{'a': 2, 'b': 3, 'c': 4}
>>> irv.eliminations_[1].rule_.gross_scores_
{'b': 5, 'c': 4}
>>> irv.eliminations_[2].rule_.gross_scores_
{'b': 9}
>>> irv.winner_
'b'
```

Remark: there exists a shortcut for the above rule in particular, the class *RuleIRV*.

By default, `propagate_tie_break` is True. So if you want to specify a tie-breaking rule, just do it in the parameters of this object, and it will also be used in the base rule. This is probably what you want to do:

```
>>> irv = RuleIteratedElimination(['a > c > b', 'b > a > c', 'c > a > b'],␣
↪weights=[1, 2, 1],
...                                base_rule=RulePlurality(), tie_break=Priority.
↪ASCENDING)
>>> irv.eliminations_[0].rule_.gross_scores_
{'a': 1, 'b': 2, 'c': 1}
>>> irv.eliminations_[1].rule_.gross_scores_
{'a': 2, 'b': 2}
>>> irv.eliminations_[2].rule_.gross_scores_
{'a': 4}
>>> irv.winner_
'a'
```

If `propagate_tie_break` is False, then there is a subtlety between the tie-breaking rule of this object, and the tie-breaking rule of the base rule. The following (somewhat contrived) example illustrates the respective roles of the two tie-breaking rules.

```
>>> rule = RuleIteratedElimination(
...     ['a', 'b', 'c', 'd', 'e'], weights=[3, 2, 2, 2, 1],
...     tie_break=Priority.DESCENDING, propagate_tie_break=False,
...     base_rule=RulePlurality(tie_break=Priority.ASCENDING),␣
↪elimination=EliminationLast(k=2))
>>> rule.eliminations_[0].rule_.gross_scores_
{'a': 3, 'b': 2, 'c': 2, 'd': 2, 'e': 1}
```

With the worst score, `e` is eliminated anyway, but we need to eliminate a second candidate because `k = 2`. In Plurality, `b`, `c` and `d` are tied, but since Plurality's tie-breaking rule is `ASCENDING`, candidates `b` or `c` get an advantage over d. Hence d is eliminated:

```
>>> rule.eliminations_[0].eliminated_
{'d', 'e'}
```

Note that the tie-breaking rule of the base rule (here Plurality) is always sufficient to compute the weak order over the candidates. This order may be finer than the elimination order, because being eliminated at the same time does not mean being tied, as d and e illustrate here:

```
>>> rule.order_
[{'a'}, {'b', 'c'}, {'d'}, {'e'}]
```

So, where does the tie-breaking rule of this object come in? It is simply used to get the strict order over the candidates, as usual in a *Rule*. In our example, since it is DESCENDING, candidate c gets an advantage over b:

```
>>> rule.strict_order_
['a', 'c', 'b', 'd', 'e']
```

**cotrailers_**

"Cotrailers" of the election, i.e. the candidates that fare worst in the election. This is the last equivalence class in order_. For example, in *RuleScoreNum*, it is the candidates that are tied for the worst score.

**Type** *NiceSet*

**cowinners_**

Cowinners of the election, i.e. the candidates that fare best in the election.. This is the first equivalence class in order_. For example, in *RuleScoreNum*, it is the candidates that are tied for the best score.

**Type** *NiceSet*

**eliminations_**

The elimination rounds. A list of *Elimination* objects. The first one corresponds to the first round, etc.

**Type** list

**n_candidates_**

Number of candidates.

**Type** int

**strict_order_**

Result of the election as a strict order over the candidates. The first element is the winner, etc. This may use the tie-breaking rule.

**Type** list

**trailer_**

The "trailer" of the election. This is the last candidate in *strict_order_* and also the unfavorable choice of the tie-breaking rule in *cotrailers_*.

**Type** object

**winner_**

The winner of the election. This is the first candidate in *strict_order_* and also the choice of the tie-breaking rule in *cowinners_*.

**Type** object

### 5.7.3 RuleScore

**class** whalrus.**RuleScore**(**args*, *tie_break: whalrus.priorities.priority.Priority = Priority.UNAMBIGUOUS*, *converter: whalrus.converters_ballot.converter_ballot.ConverterBallot = None*, ***kwargs*)

A voting rule with scores (which are not necessarily numbers).

Each candidate is assigned a score (not necessarily a number), and the the cowinners are the candidates with the best score, in the sense defined by `compare_scores()`.

**best_score_**

The best score.

**Type** object

**compare_scores**(*one: object*, *another: object*) → int

Compare two scores.

**Parameters**

- **one** (*object*) – A score.
- **another** (*object*) – A score.

**Returns** 0 if they are equal, a positive number if `one` is greater than `another`, a negative number otherwise.

**Return type** int

**cotrailers_**

"Cotrailers". The set of candidates with the worst score.

**Type** *NiceSet*

**cowinners_**

Cowinners. The set of candidates with the best score.

**Type** *NiceSet*

**n_candidates_**

Number of candidates.

**Type** int

**order_**

Result of the election as a (weak) order over the candidates. It is a list of `NiceSet`. The first set contains the candidates that have the best score, the second set contains those with the second best score, etc.

**Type** list

**scores_**

The scores. To each candidate, this dictionary assigns a score (non necessarily a number).

**Type** *NiceDict*

**strict_order_**

Result of the election as a strict order over the candidates. The first element is the winner, etc. This may use the tie-breaking rule.

**Type** list

**trailer_**

The "trailer" of the election. This is the last candidate in `strict_order_` and also the unfavorable choice of the tie-breaking rule in `cotrailers_`.

**Type** object

**winner_**

The winner of the election. This is the first candidate in `strict_order_` and also the choice of the tie-breaking rule in `cowinners_`.

**Type** object

**worst_score_**

The worst score.

**Type** object

### 5.7.4 RuleScoreNum

**class** whalrus.**RuleScoreNum**(**args*, *tie_break: whalrus.priorities.priority.Priority = Priority.UNAMBIGUOUS*, *converter: whalrus.converters_ballot.converter_ballot.ConverterBallot = None*, ***kwargs*)

A voting rule with numeric scores.

This is a voting rule where each candidate is assigned a numeric score, and the candidates with the best score are declared the cowinners.

**average_score_**

The average score.

**Type** Number

**average_score_as_float_**

The average score as a float. It is the same as `average_score_`, but converted to a float.

**Type** float

**best_score_as_float_**

The best score as a float. It is the same as `RuleScore.best_score_`, but converted to a float.

**Type** float

**compare_scores**(*one: numbers.Number*, *another: numbers.Number*) → int

Compare two scores.

**Parameters**

- **one** (*object*) – A score.
- **another** (*object*) – A score.

**Returns** 0 if they are equal, a positive number if `one` is greater than `another`, a negative number otherwise.

**Return type** int

**cotrailers_**

"Cotrailers". The set of candidates with the worst score.

**Type** *NiceSet*

**cowinners_**

Cowinners. The set of candidates with the best score.

**Type** *NiceSet*

**n_candidates_**

Number of candidates.

**Type** int

**scores_**

The scores. To each candidate, this dictionary assigns a numeric score.

**Type** *NiceDict*

**scores_as_floats_**

Scores as floats. It is the same as `scores_`, but converted to floats.

**Type** *NiceDict*

**strict_order_**

Result of the election as a strict order over the candidates. The first element is the winner, etc. This may use the tie-breaking rule.

**Type** list

**trailer_**

The "trailer" of the election. This is the last candidate in `strict_order_` and also the unfavorable choice of the tie-breaking rule in `cotrailers_`.

**Type** object

**winner_**

The winner of the election. This is the first candidate in `strict_order_` and also the choice of the tie-breaking rule in `cowinners_`.

**Type** object

**worst_score_as_float_**

The worst score as a float. It is the same as `RuleScore.worst_score_`, but converted to a float.

**Type** float

### 5.7.5 RuleScoreNumAverage

**class** whalrus.**RuleScoreNumAverage**(**args*, *scorer: whalrus.scorers.scorer.Scorer = None*, *default_average: numbers.Number = 0*, ***kwargs*)

A voting rule where each candidate's score is an average of the scores provided by the ballots.

**Parameters**

- **args** – Cf. parent class.
- **scorer** (`Scorer`) – For each ballot, it is in charge of computing its contribution to each candidate's score.
- **default_average** (*Number*) – The default average score of a candidate when it receives no score whatsoever. It may happen, for example, if all voters abstain about this candidate. This avoids a division by zero when computing this candidate's average score.
- **kwargs** – Cf. parent class.

#### Examples

Cf. `RuleRangeVoting` for some examples.

**average_score_**

The average score.

**Type** Number

**average_score_as_float_**

The average score as a float. It is the same as *average_score_*, but converted to a float.

**Type** float

**best_score_as_float_**

The best score as a float. It is the same as *RuleScore.best_score_*, but converted to a float.

**Type** float

**compare_scores**(*one: numbers.Number*, *another: numbers.Number*) → int

Compare two scores.

**Parameters**

- **one** (*object*) – A score.
- **another** (*object*) – A score.

**Returns** 0 if they are equal, a positive number if one is greater than another, a negative number otherwise.

**Return type** int

**cotrailers_**

"Cotrailers". The set of candidates with the worst score.

**Type** *NiceSet*

**cowinners_**

Cowinners. The set of candidates with the best score.

**Type** *NiceSet*

**gross_scores_**

The gross scores of the candidates. For each candidate, this dictionary gives the sum of its scores, multiplied by the weights of the corresponding voters. This is the numerator in the candidate's average score.

**Type** *NiceDict*

**gross_scores_as_floats_**

Gross scores as floats. It is the same as *gross_scores_*, but converted to floats.

**Type** *NiceDict*

**n_candidates_**

Number of candidates.

**Type** int

**scores_as_floats_**

Scores as floats. It is the same as scores_, but converted to floats.

**Type** *NiceDict*

**strict_order_**

Result of the election as a strict order over the candidates. The first element is the winner, etc. This may use the tie-breaking rule.

**Type** list

**trailer_**

The "trailer" of the election. This is the last candidate in *strict_order_* and also the unfavorable choice of the tie-breaking rule in *cotrailers_*.

**Type** object

**weights_**

The weights used for the candidates. For each candidate, this dictionary gives the total weight for this candidate, i.e. the total weight of all voters who assign a score to this candidate. This is the denominator in the candidate's average score.

**Type** *NiceDict*

**weights_as_floats_**

Weights as floats. It is the same as *weights_*, but converted to floats.

**Type** *NiceDict*

**winner_**

The winner of the election. This is the first candidate in *strict_order_* and also the choice of the tie-breaking rule in *cowinners_*.

**Type** object

**worst_score_as_float_**

The worst score as a float. It is the same as *RuleScore.worst_score_*, but converted to a float.

**Type** float

### 5.7.6 RuleScoreNumRowSum

**class** whalrus.**RuleScoreNumRowSum**(**args*, *matrix: whalrus.matrices.matrix.Matrix = None*, ***kwargs*)

Rule where the winner is the candidate having the highest row sum in some matrix.

The score of a candidate is the sum of the non-diagonal elements of its row in *matrix_*.

**Parameters**

- **args** – Cf. parent class.
- **matrix** (Matrix) – The matrix upon which the scores are based.
- **kwargs** – Cf. parent class.

**average_score_**

The average score.

**Type** Number

**average_score_as_float_**

The average score as a float. It is the same as *average_score_*, but converted to a float.

**Type** float

**best_score_as_float_**

The best score as a float. It is the same as *RuleScore.best_score_*, but converted to a float.

**Type** float

**compare_scores**(*one: numbers.Number*, *another: numbers.Number*) → int

Compare two scores.

**Parameters**

- **one** (*object*) – A score.
- **another** (*object*) – A score.

**Returns** 0 if they are equal, a positive number if `one` is greater than `another`, a negative number otherwise.

**Return type** int

**cotrailers_**

“Cotrailers”. The set of candidates with the worst score.

**Type** *NiceSet*

**cowinners_**

Cowinners. The set of candidates with the best score.

**Type** *NiceSet*

**matrix_**

The matrix (once computed with the given profile).

**Type** *Matrix*

**n_candidates_**

Number of candidates.

**Type** int

**scores_as_floats_**

Scores as floats. It is the same as `scores_`, but converted to floats.

**Type** *NiceDict*

**strict_order_**

Result of the election as a strict order over the candidates. The first element is the winner, etc. This may use the tie-breaking rule.

**Type** list

**trailer_**

The “trailer” of the election. This is the last candidate in `strict_order_` and also the unfavorable choice of the tie-breaking rule in `cotrailers_`.

**Type** object

**winner_**

The winner of the election. This is the first candidate in `strict_order_` and also the choice of the tie-breaking rule in `cowinners_`.

**Type** object

**worst_score_as_float_**

The worst score as a float. It is the same as `RuleScore.worst_score_`, but converted to a float.

**Type** float

### 5.7.7 RuleScorePositional

**class** whalrus.**RuleScorePositional**(**args, converter: whalrus.converters_ballot.converter_ballot.ConverterBallot = None, points_scheme: list = None, **kwargs*)

A positional scoring rule.

**Parameters**

- **args** – Cf. parent class.
- **converter** (ConverterBallot) – Default: *ConverterBallotToStrictOrder*.
- **points_scheme** (*list*) – The list of points to be attributed to the candidates of a ballot. Cf. *ScorerPositional*.
- **kwargs** – Cf. parent class.

### Examples

```
>>> RuleScorePositional(['a > b > c', 'b > c > a'], points_scheme=[3, 2, 1]).
↪gross_scores_
{'a': 4, 'b': 5, 'c': 3}
```

Since this voting rule needs strict orders, problems may occur as soon as there is indifference in the ballots. To avoid these issues, specify the ballot converter explicitly:

```
>>> RuleScorePositional(['a > b ~ c', 'b > c > a'], points_scheme=[1, 1, 0],
...     converter=ConverterBallotToStrictOrder(priority=Priority.ASCENDING)).
↪gross_scores_
{'a': 1, 'b': 2, 'c': 1}
```

**average_score_**

The average score.

**Type** Number

**average_score_as_float_**

The average score as a float. It is the same as *average_score_*, but converted to a float.

**Type** float

**best_score_as_float_**

The best score as a float. It is the same as *RuleScore.best_score_*, but converted to a float.

**Type** float

**compare_scores** (*one: numbers.Number*, *another: numbers.Number*) → int

Compare two scores.

**Parameters**

- **one** (*object*) – A score.
- **another** (*object*) – A score.

**Returns** 0 if they are equal, a positive number if `one` is greater than `another`, a negative number otherwise.

**Return type** int

**cotrailers_**

"Cotrailers". The set of candidates with the worst score.

**Type** *NiceSet*

**cowinners_**

Cowinners. The set of candidates with the best score.

**Type** *NiceSet*

**gross_scores_**

The gross scores of the candidates. For each candidate, this dictionary gives the sum of its scores, multiplied by the weights of the corresponding voters. This is the numerator in the candidate's average score.

**Type** *NiceDict*

**gross_scores_as_floats_**

Gross scores as floats. It is the same as *gross_scores_*, but converted to floats.

**Type** *NiceDict*

**n_candidates_**

Number of candidates.

**Type** int

**scores_as_floats_**

Scores as floats. It is the same as `scores_`, but converted to floats.

**Type** *NiceDict*

**strict_order_**

Result of the election as a strict order over the candidates. The first element is the winner, etc. This may use the tie-breaking rule.

**Type** list

**trailer_**

The "trailer" of the election. This is the last candidate in *strict_order_* and also the unfavorable choice of the tie-breaking rule in *cotrailers_*.

**Type** object

**weights_**

The weights used for the candidates. For each candidate, this dictionary gives the total weight for this candidate, i.e. the total weight of all voters who assign a score to this candidate. This is the denominator in the candidate's average score.

**Type** *NiceDict*

**weights_as_floats_**

Weights as floats. It is the same as *weights_*, but converted to floats.

**Type** *NiceDict*

**winner_**

The winner of the election. This is the first candidate in *strict_order_* and also the choice of the tie-breaking rule in *cowinners_*.

**Type** object

**worst_score_as_float_**

The worst score as a float. It is the same as *RuleScore.worst_score_*, but converted to a float.

**Type** float

### 5.7.8 RuleSequentialElimination

**class** whalrus.**RuleSequentialElimination**(**args, rules: Union[list, whalrus.rules.rule.Rule] = None, eliminations: Union[list, whalrus.eliminations.elimination.Elimination] = None, propagate_tie_break=True, **kwargs*)

A rule by sequential elimination (such as *RuleTwoRound*).

**Parameters**

- **`args`** – Cf. parent class.
- **`rules`** (*`list of Rule`*) – A list of rules, one for each round. Unlike for *`RuleIteratedElimination`*, different rounds may use different voting rules.
- **`eliminations`** (*`list of Elimination`*) – A list of elimination algorithms, one for each round except the last one.
- **`propagate_tie_break`** (*`bool`*) – If True (default), then the tie-breaking rule of this object is also used for the base rules. Cf. *`RuleIteratedElimination`* for more explanation on this parameter.
- **`kwargs`** – Cf. parent class.

## Examples

```
>>> rule = RuleSequentialElimination(
...     ['a > b > c > d > e', 'b > c > d > e > a'], weights=[2, 1],
...     rules=[RuleBorda(), RulePlurality(), RulePlurality()],
...     eliminations=[EliminationBelowAverage(), EliminationLast(k=1)])
>>> rule.elimination_rounds_[0].rule_.gross_scores_
{'a': 8, 'b': 10, 'c': 7, 'd': 4, 'e': 1}
>>> rule.elimination_rounds_[1].rule_.gross_scores_
{'a': 2, 'b': 1, 'c': 0}
>>> rule.final_round_.gross_scores_
{'a': 2, 'b': 1}
```

If `rules` is not a list, the number of rounds is inferred from `eliminations`. An application of this is to define the two-round system:

```
>>> rule = RuleSequentialElimination(
...     ['a > b > c > d > e', 'b > a > c > d > e', 'c > a > b > d > e'],␣
↪weights=[2, 2, 1],
...     rules=RulePlurality(), eliminations=[EliminationLast(k=-2)])
>>> rule.elimination_rounds_[0].rule_.gross_scores_
{'a': 2, 'b': 2, 'c': 1, 'd': 0, 'e': 0}
>>> rule.final_round_.gross_scores_
{'a': 3, 'b': 2}
```

Note: there exists a shortcut for the above rule in particular, the class *`RuleTwoRound`*.

Similarly, if `elimination` is not a list, the number of rounds is deduced from `rules`:

```
>>> rule = RuleSequentialElimination(
...     ['a > b > c > d > e', 'b > a > c > d > e'], weights=[2, 1],
...     rules=[RuleBorda(), RuleBorda(), RulePlurality()],␣
↪eliminations=EliminationLast(k=1))
>>> rule.elimination_rounds_[0].rule_.gross_scores_
{'a': 11, 'b': 10, 'c': 6, 'd': 3, 'e': 0}
>>> rule.elimination_rounds_[1].rule_.gross_scores_
{'a': 8, 'b': 7, 'c': 3, 'd': 0}
>>> rule.final_round_.gross_scores_
{'a': 2, 'b': 1, 'c': 0}
```

**`cotrailers_`**

“Cotrailers” of the election, i.e. the candidates that fare worst in the election. This is the last equivalence class in `order_`. For example, in *`RuleScoreNum`*, it is the candidates that are tied for the worst score.

**Type** *NiceSet*

**cowinners_**

Cowinners of the election, i.e. the candidates that fare best in the election.. This is the first equivalence class in order_. For example, in *RuleScoreNum*, it is the candidates that are tied for the best score.

**Type** *NiceSet*

**elimination_rounds_**

The elimination rounds. A list of *Elimination* objects. All rounds except the last one.

**Type** list

**final_round_**

The final round, which decides the winner of the election.

**Type** *Rule*

**n_candidates_**

Number of candidates.

**Type** int

**rounds_**

The rounds. All rounds but the last one are *Elimination* objects. The last one is a *Rule* object.

### Examples

Note that in some cases, there may be fewer actual rounds than declared in the definition of the rule:

```
>>> rule = RuleSequentialElimination(
...     ['a > b > c > d', 'a > c > d > b', 'a > d > b > c'],
...     rules=[RuleBorda(), RulePlurality(), RulePlurality()],
...     eliminations=[EliminationBelowAverage(), EliminationLast(k=1)])
>>> len(rule.rounds_)
2
>>> rule.elimination_rounds_[0].rule_.gross_scores_
{'a': 9, 'b': 3, 'c': 3, 'd': 3}
>>> rule.final_round_.gross_scores_
{'a': 3}
```

**Type** list

**strict_order_**

Result of the election as a strict order over the candidates. The first element is the winner, etc. This may use the tie-breaking rule.

**Type** list

**trailer_**

The "trailer" of the election. This is the last candidate in *strict_order_* and also the unfavorable choice of the tie-breaking rule in *cotrailers_*.

**Type** object

**winner_**

The winner of the election. This is the first candidate in *strict_order_* and also the choice of the tie-breaking rule in *cowinners_*.

**Type** object

### 5.7.9 RuleSequentialTieBreak

**class** whalrus.**RuleSequentialTieBreak**(**args, rules: list = None, **kwargs*)

A rule by sequential tie-break.

**Parameters**

- **args** – Cf. parent class.
- **rules** (*list of Rule*) –
- **kwargs** – Cf. parent class.

#### Examples

The winner is determined by the first rule. If there is a tie, it is broken by the second rule. Etc. There may still be a tie at the end: in that case, it is broken by the tie-breaking rule of this object.

```
>>> rule = RuleSequentialTieBreak(
...     ['a > d > e > b > c', 'b > d > e > a > c', 'c > d > e > a > b',
...      'd > e > b > a > c', 'e > d > b > a > c'],
...     weights=[2, 2, 2, 1, 1],
...     rules=[RulePlurality(), RuleBorda()], tie_break=Priority.ASCENDING)
>>> rule.rules_[0].gross_scores_
{'a': 2, 'b': 2, 'c': 2, 'd': 1, 'e': 1}
>>> rule.rules_[1].gross_scores_
{'a': 14, 'b': 14, 'c': 8, 'd': 25, 'e': 19}
>>> rule.order_
[{'a', 'b'}, {'c'}, {'d'}, {'e'}]
>>> rule.winner_
'a'
```

**cotrailers_**

"Cotrailers" of the election, i.e. the candidates that fare worst in the election. This is the last equivalence class in order_. For example, in *RuleScoreNum*, it is the candidates that are tied for the worst score.

**Type** *NiceSet*

**cowinners_**

Cowinners of the election, i.e. the candidates that fare best in the election.. This is the first equivalence class in order_. For example, in *RuleScoreNum*, it is the candidates that are tied for the best score.

**Type** *NiceSet*

**n_candidates_**

Number of candidates.

**Type** int

**rules_**

The rules (once applied to the profile).

**Type** list

**strict_order_**

Result of the election as a strict order over the candidates. The first element is the winner, etc. This may use the tie-breaking rule.

**Type** list

**trailer_**

The "trailer" of the election. This is the last candidate in *strict_order_* and also the unfavorable choice of the tie-breaking rule in *cotrailers_*.

**Type** object

**winner_**

The winner of the election. This is the first candidate in *strict_order_* and also the choice of the tie-breaking rule in *cowinners_*.

**Type** object

## 5.8 Rule: In Particular

### 5.8.1 RuleApproval

**class** whalrus.**RuleApproval**(**args*, *converter: whalrus.converters_ballot.converter_ballot.ConverterBallot = None*, ***kwargs*)

Approval voting.

**Parameters**

- **args** – Cf. parent class.
- **converter** (ConverterBallot) – Default: `ConverterBallotToGrades(scale=ScaleRange(0, 1))`. This is the only difference with the parent class *RuleRangeVoting*.
- **kwargs** – Cf. parent class.

#### Examples

```
>>> RuleApproval([{'a': 1, 'b': 0, 'c': 0}, {'a': 1, 'b': 1, 'c': 0}]).gross_
↪scores_
{'a': 2, 'b': 1, 'c': 0}
```

**average_score_**

The average score.

**Type** Number

**average_score_as_float_**

The average score as a float. It is the same as *average_score_*, but converted to a float.

**Type** float

**best_score_as_float_**

The best score as a float. It is the same as *RuleScore.best_score_*, but converted to a float.

**Type** float

**compare_scores**(*one: numbers.Number*, *another: numbers.Number*) → int

Compare two scores.

**Parameters**

- **one** (*object*) – A score.
- **another** (*object*) – A score.

**Returns** 0 if they are equal, a positive number if `one` is greater than `another`, a negative number otherwise.

**Return type** int

**`cotrailers_`**

"Cotrailers". The set of candidates with the worst score.

**Type** *NiceSet*

**`cowinners_`**

Cowinners. The set of candidates with the best score.

**Type** *NiceSet*

**`gross_scores_`**

The gross scores of the candidates. For each candidate, this dictionary gives the sum of its scores, multiplied by the weights of the corresponding voters. This is the numerator in the candidate's average score.

**Type** *NiceDict*

**`gross_scores_as_floats_`**

Gross scores as floats. It is the same as `gross_scores_`, but converted to floats.

**Type** *NiceDict*

**`n_candidates_`**

Number of candidates.

**Type** int

**`scores_as_floats_`**

Scores as floats. It is the same as `scores_`, but converted to floats.

**Type** *NiceDict*

**`strict_order_`**

Result of the election as a strict order over the candidates. The first element is the winner, etc. This may use the tie-breaking rule.

**Type** list

**`trailer_`**

The "trailer" of the election. This is the last candidate in `strict_order_` and also the unfavorable choice of the tie-breaking rule in `cotrailers_`.

**Type** object

**`weights_`**

The weights used for the candidates. For each candidate, this dictionary gives the total weight for this candidate, i.e. the total weight of all voters who assign a score to this candidate. This is the denominator in the candidate's average score.

**Type** *NiceDict*

**`weights_as_floats_`**

Weights as floats. It is the same as `weights_`, but converted to floats.

**Type** *NiceDict*

**`winner_`**

The winner of the election. This is the first candidate in `strict_order_` and also the choice of the tie-breaking rule in `cowinners_`.

**Type** object

**worst_score_as_float_**

The worst score as a float. It is the same as *RuleScore.worst_score_*, but converted to a float.

**Type** float

## 5.8.2 RuleBaldwin

**class** whalrus.**RuleBaldwin**(**args*, *base_rule: whalrus.rules.rule.Rule = None*, *elimination: whalrus.eliminations.elimination.Elimination = None*, ***kwargs*)

Baldwin's rule.

At each round, the candidate with the lowest Borda score is eliminated.

**Parameters**

- **args** – Cf. parent class.
- **base_rule** (Rule) – Default: *RuleBorda*.
- **elimination** (Elimination) – Default: *EliminationLast* with k=1.
- **kwargs** – Cf. parent class.

### Examples

```
>>> rule = RuleBaldwin(['a > b > c', 'a > b ~ c'])
>>> rule.eliminations_[0].rule_.gross_scores_
{'a': 4, 'b': Fraction(3, 2), 'c': Fraction(1, 2)}
>>> rule.eliminations_[1].rule_.gross_scores_
{'a': 2, 'b': 0}
>>> rule.eliminations_[2].rule_.gross_scores_
{'a': 0}
>>> rule.winner_
'a'
```

**cotrailers_**

"Cotrailers" of the election, i.e. the candidates that fare worst in the election. This is the last equivalence class in order_. For example, in *RuleScoreNum*, it is the candidates that are tied for the worst score.

**Type** *NiceSet*

**cowinners_**

Cowinners of the election, i.e. the candidates that fare best in the election.. This is the first equivalence class in order_. For example, in *RuleScoreNum*, it is the candidates that are tied for the best score.

**Type** *NiceSet*

**eliminations_**

The elimination rounds. A list of *Elimination* objects. The first one corresponds to the first round, etc.

**Type** list

**n_candidates_**

Number of candidates.

**Type** int

**strict_order_**

Result of the election as a strict order over the candidates. The first element is the winner, etc. This may use the tie-breaking rule.

**Type** list

**trailer_**

The "trailer" of the election. This is the last candidate in *strict_order_* and also the unfavorable choice of the tie-breaking rule in *cotrailers_*.

**Type** object

**winner_**

The winner of the election. This is the first candidate in *strict_order_* and also the choice of the tie-breaking rule in *cowinners_*.

**Type** object

### 5.8.3 RuleBlack

**class** whalrus.**RuleBlack**(**args*, *rule_condorcet: whalrus.rules.rule.Rule = None*, *rule_borda: whalrus.rules.rule.Rule = None*, ***kwargs*)

Black's rule.

**Parameters**

- **args** – Cf. parent class.
- **rule_condorcet** (Rule) – Used as the main victory criterion. Default: *RuleCondorcet*.
- **rule_borda** (Rule) – Used as the secondary victory criterion. Default: *RuleBorda*.
- **kwargs** – Cf. parent class.

#### Examples

As a main victory criterion, the Condorcet winner is elected (even if it does not have the highest Borda score):

```
>>> rule = RuleBlack(ballots=['a > b > c', 'b > c > a'], weights=[3, 2])
>>> rule.rule_condorcet_.matrix_majority_.matrix_weighted_majority_.as_array_
array([[0, Fraction(3, 5), Fraction(3, 5)],
       [Fraction(2, 5), 0, 1],
       [Fraction(2, 5), 0, 0]], dtype=object)
>>> rule.order_
[{'a'}, {'b'}, {'c'}]
```

When there is no Condorcet winner, candidates are sorted according to their Borda scores:

```
>>> rule = RuleBlack(ballots=['a > b > c', 'b > c > a', 'c > a > b'], weights=[3,␣
↪2, 2])
>>> rule.rule_condorcet_.matrix_majority_.matrix_weighted_majority_.as_array_
array([[0, Fraction(5, 7), Fraction(3, 7)],
       [Fraction(2, 7), 0, Fraction(5, 7)],
       [Fraction(4, 7), Fraction(2, 7), 0]], dtype=object)
>>> rule.order_
[{'a'}, {'b'}, {'c'}]
```

**cotrailers_**

"Cotrailers" of the election, i.e. the candidates that fare worst in the election. This is the last equivalence class in order_. For example, in *RuleScoreNum*, it is the candidates that are tied for the worst score.

**Type** *NiceSet*

**cowinners_**

Cowinners of the election, i.e. the candidates that fare best in the election.. This is the first equivalence class in `order_`. For example, in *RuleScoreNum*, it is the candidates that are tied for the best score.

**Type** *NiceSet*

**n_candidates_**

Number of candidates.

**Type** int

**rule_borda_**

The Borda rule (once applied to the profile).

### Examples

```
>>> rule = RuleBlack(ballots=['a > b > c', 'b > c > a'], weights=[3, 2])
>>> rule.rule_borda_.scores_
{'a': Fraction(6, 5), 'b': Fraction(7, 5), 'c': Fraction(2, 5)}
```

**Type** *Rule*

**rule_condorcet_**

The Condorcet rule (once applied to the profile).

### Examples

```
>>> rule = RuleBlack(ballots=['a > b > c', 'b > c > a'], weights=[3, 2])
>>> rule.rule_condorcet_.matrix_majority_.as_array_
array([[Fraction(1, 2), 1, 1],
       [0, Fraction(1, 2), 1],
       [0, 0, Fraction(1, 2)]], dtype=object)
```

**Type** *Rule*

**rules_**

The rules (once applied to the profile).

**Type** list

**strict_order_**

Result of the election as a strict order over the candidates. The first element is the winner, etc. This may use the tie-breaking rule.

**Type** list

**trailer_**

The "trailer" of the election. This is the last candidate in *strict_order_* and also the unfavorable choice of the tie-breaking rule in *cotrailers_*.

**Type** object

**winner_**

The winner of the election. This is the first candidate in *strict_order_* and also the choice of the tie-breaking rule in *cowinners_*.

**Type** object

### 5.8.4 RuleBorda

**class** whalrus.**RuleBorda**(**args, converter: whalrus.converters_ballot.converter_ballot.ConverterBallot = None, scorer: whalrus.scorers.scorer.Scorer = None, **kwargs*)

The Borda rule.

**Parameters**

- **args** – Cf. parent class.
- **converter** (ConverterBallot) – Default: *ConverterBallotToOrder*.
- **scorer** (Scorer) – Default: *ScorerBorda*.
- **kwargs** – Cf. parent class.

#### Examples

```
>>> rule = RuleBorda(['a ~ b > c', 'b > c > a'])
>>> rule.gross_scores_
{'a': Fraction(3, 2), 'b': Fraction(7, 2), 'c': 1}
>>> rule.scores_
{'a': Fraction(3, 4), 'b': Fraction(7, 4), 'c': Fraction(1, 2)}
```

**average_score_**

The average score.

**Type** Number

**average_score_as_float_**

The average score as a float. It is the same as *average_score_*, but converted to a float.

**Type** float

**best_score_as_float_**

The best score as a float. It is the same as *RuleScore.best_score_*, but converted to a float.

**Type** float

**compare_scores**(*one: numbers.Number, another: numbers.Number*) → int

Compare two scores.

**Parameters**

- **one** (*object*) – A score.
- **another** (*object*) – A score.

**Returns** 0 if they are equal, a positive number if one is greater than another, a negative number otherwise.

**Return type** int

**cotrailers_**

"Cotrailers". The set of candidates with the worst score.

**Type** *NiceSet*

**cowinners_**

Cowinners. The set of candidates with the best score.

**Type** *NiceSet*

**gross_scores_**

The gross scores of the candidates. For each candidate, this dictionary gives the sum of its scores, multiplied by the weights of the corresponding voters. This is the numerator in the candidate's average score.

**Type** *NiceDict*

**gross_scores_as_floats_**

Gross scores as floats. It is the same as `gross_scores_`, but converted to floats.

**Type** *NiceDict*

**n_candidates_**

Number of candidates.

**Type** int

**scores_as_floats_**

Scores as floats. It is the same as `scores_`, but converted to floats.

**Type** *NiceDict*

**strict_order_**

Result of the election as a strict order over the candidates. The first element is the winner, etc. This may use the tie-breaking rule.

**Type** list

**trailer_**

The "trailer" of the election. This is the last candidate in `strict_order_` and also the unfavorable choice of the tie-breaking rule in `cotrailers_`.

**Type** object

**weights_**

The weights used for the candidates. For each candidate, this dictionary gives the total weight for this candidate, i.e. the total weight of all voters who assign a score to this candidate. This is the denominator in the candidate's average score.

**Type** *NiceDict*

**weights_as_floats_**

Weights as floats. It is the same as `weights_`, but converted to floats.

**Type** *NiceDict*

**winner_**

The winner of the election. This is the first candidate in `strict_order_` and also the choice of the tie-breaking rule in `cowinners_`.

**Type** object

**worst_score_as_float_**

The worst score as a float. It is the same as `RuleScore.worst_score_`, but converted to a float.

**Type** float

### 5.8.5 RuleBucklinByRounds

**class** whalrus.**RuleBucklinByRounds**(**args*, *converter: whalrus.converters_ballot.converter_ballot.ConverterBallot = None*, *scorer: whalrus.scorers.scorer_bucklin.ScorerBucklin = None*, ***kwargs*)

Bucklin's rule (round by round version).

During the first round, a candidate's score is the proportion of voters who rank it first. During the second round, its score is the proportion of voters who rank it first or second. Etc. More precisely, at each round, the `scorer` is used with `k` equal to the round number; cf. *ScorerBucklin*.

For another variant of Bucklin's rule, cf. *RuleBucklinInstant*.

**Parameters**

- **args** – Cf. parent class.
- **converter** (ConverterBallot) – Default: *ConverterBallotToOrder*.
- **scorer** (Scorer) – Default: *ScorerBucklin*.
- **kwargs** – Cf. parent class.

#### Examples

```
>>> rule = RuleBucklinByRounds(['a > b > c > d', 'b > a > c > d',
...                             'c > a > b > d', 'd > a > b > c'])
>>> rule.detailed_scores_[0]
{'a': Fraction(1, 4), 'b': Fraction(1, 4), 'c': Fraction(1, 4), 'd': Fraction(1,␣
↪4)}
>>> rule.detailed_scores_[1]
{'a': 1, 'b': Fraction(1, 2), 'c': Fraction(1, 4), 'd': Fraction(1, 4)}
>>> rule.n_rounds_
2
>>> rule.scores_
{'a': 1, 'b': Fraction(1, 2), 'c': Fraction(1, 4), 'd': Fraction(1, 4)}
>>> rule.winner_
'a'
```

**average_score_**

The average score.

**Type** Number

**average_score_as_float_**

The average score as a float. It is the same as *average_score_*, but converted to a float.

**Type** float

**best_score_as_float_**

The best score as a float. It is the same as *RuleScore.best_score_*, but converted to a float.

**Type** float

**compare_scores**(*one: numbers.Number*, *another: numbers.Number*) → int

Compare two scores.

**Parameters**

- **one** (*object*) – A score.

- **another** (*object*) – A score.

**Returns** 0 if they are equal, a positive number if `one` is greater than `another`, a negative number otherwise.

**Return type** int

**cotrailers_**

"Cotrailers". The set of candidates with the worst score.

**Type** *NiceSet*

**cowinners_**

Cowinners. The set of candidates with the best score.

**Type** *NiceSet*

**detailed_scores_**

Detailed scores. A list of `NiceDict`. The first dictionary gives the scores of the first round, etc.

**Type** list

**detailed_scores_as_floats_**

Detailed scores, as floats. It is the same as *`detailed_scores_`*, but converted to floats.

### Examples

```
>>> rule = RuleBucklinByRounds(['a > b > c > d', 'b > a > c > d',
...                             'c > a > b > d', 'd > a > b > c'])
>>> rule.detailed_scores_as_floats_[0]
{'a': 0.25, 'b': 0.25, 'c': 0.25, 'd': 0.25}
>>> rule.detailed_scores_as_floats_[1]
{'a': 1.0, 'b': 0.5, 'c': 0.25, 'd': 0.25}
```

**Type** list

**n_candidates_**

Number of candidates.

**Type** int

**n_rounds_**

The number of rounds.

**Type** int

**scores_**

The scores. For each candidate, it gives its score during the final round, i.e. the first round where at least one candidate has a score above 1 / 2.

**Type** *NiceDict*

**scores_as_floats_**

Scores as floats. It is the same as *`scores_`*, but converted to floats.

**Type** *NiceDict*

**strict_order_**

Result of the election as a strict order over the candidates. The first element is the winner, etc. This may use the tie-breaking rule.

**Type** list

**trailer_**

The "trailer" of the election. This is the last candidate in *strict_order_* and also the unfavorable choice of the tie-breaking rule in *cotrailers_*.

**Type** object

**winner_**

The winner of the election. This is the first candidate in *strict_order_* and also the choice of the tie-breaking rule in *cowinners_*.

**Type** object

**worst_score_as_float_**

The worst score as a float. It is the same as *RuleScore.worst_score_*, but converted to a float.

**Type** float

### 5.8.6 RuleBucklinInstant

**class** whalrus.**RuleBucklinInstant**(**args, converter: whalrus.converters_ballot.converter_ballot.ConverterBallot = None, scorer: whalrus.scorers.scorer.Scorer = None, default_median: object = 0, **kwargs*)

Bucklin's rule (instant version).

For each candidate, its median Borda score *m* is computed. Let *x* be the number of voters who give this candidate a Borda score that is greater or equal to *m*. Then the candidate's score is *(m, x)*. Scores are compared lexicographically.

When preferences are strict orders, it is equivalent to say that:

- The candidate with the lowest median rank is declared the winner.
- If several candidates have the lowest median rank, this tie is broken by examining how many voters rank each of them with this rank or better.

For another variant of Bucklin's rule, cf. *RuleBucklinByRounds*.

**Parameters**

- **args** – Cf. parent class.
- **converter** (ConverterBallot) – Default: *ConverterBallotToOrder*.
- **scorer** (Scorer) – Default: *ScorerBorda* with absent_give_points=True, absent_receive_points=None, unordered_give_points=True, unordered_receive_points=False.
- **default_median** (*object*) – The default median of a candidate when it receives no score whatsoever.
- **kwargs** – Cf. parent class.

#### Examples

```
>>> rule = RuleBucklinInstant(ballots=['a > b > c', 'b > a > c', 'c > a > b'])
>>> rule.scores_
{'a': (1, 3), 'b': (1, 2), 'c': (0, 3)}
>>> rule.winner_
'a'
```

With the default settings, and when preferences are strict total orders, *RuleBucklinByRounds* and *RuleBucklinInstant* have the same winner (although not necessarily the same order over the candidates). Otherwise, the winners may differ:

```
>>> profile = Profile(ballots=['a > b > c > d', 'b > a ~ d > c', 'c > a ~ d > b'],
...                   weights=[3, 3, 4])
>>> rule_bucklin_by_rounds = RuleBucklinByRounds(profile)
>>> rule_bucklin_by_rounds.detailed_scores_[0]
{'a': Fraction(3, 10), 'b': Fraction(3, 10), 'c': Fraction(2, 5), 'd': 0}
>>> rule_bucklin_by_rounds.detailed_scores_[1]
{'a': Fraction(13, 20), 'b': Fraction(3, 5), 'c': Fraction(2, 5), 'd': Fraction(7,
↪ 20)}
>>> rule_bucklin_by_rounds.winner_
'a'
>>> rule_bucklin_instant = RuleBucklinInstant(profile)
>>> rule_bucklin_instant.scores_
{'a': (Fraction(3, 2), 10), 'b': (2, 6), 'c': (1, 7), 'd': (Fraction(3, 2), 7)}
>>> RuleBucklinInstant(profile).winner_
'b'
```

**best_score_**

The best score.

**Type** object

**compare_scores** (*one: tuple*, *another: tuple*) → int

Compare two scores.

**Parameters**

- **one** (*object*) – A score.
- **another** (*object*) – A score.

**Returns** 0 if they are equal, a positive number if `one` is greater than `another`, a negative number otherwise.

**Return type** int

**cotrailers_**

"Cotrailers". The set of candidates with the worst score.

**Type** *NiceSet*

**cowinners_**

Cowinners. The set of candidates with the best score.

**Type** *NiceSet*

**n_candidates_**

Number of candidates.

**Type** int

**order_**

Result of the election as a (weak) order over the candidates. It is a list of `NiceSet`. The first set contains the candidates that have the best score, the second set contains those with the second best score, etc.

**Type** list

**scores_as_floats_**

Scores as floats. It is the same as `scores_`, but converted to floats.

### Examples

```
>>> rule = RuleBucklinInstant(ballots=['a > b > c', 'b > a > c', 'c > a > b'])
>>> rule.scores_as_floats_
{'a': (1.0, 3.0), 'b': (1.0, 2.0), 'c': (0.0, 3.0)}
```

**Type** *NiceDict*

**strict_order_**

Result of the election as a strict order over the candidates. The first element is the winner, etc. This may use the tie-breaking rule.

**Type** list

**trailer_**

The "trailer" of the election. This is the last candidate in *strict_order_* and also the unfavorable choice of the tie-breaking rule in *cotrailers_*.

**Type** object

**winner_**

The winner of the election. This is the first candidate in *strict_order_* and also the choice of the tie-breaking rule in *cowinners_*.

**Type** object

**worst_score_**

The worst score.

**Type** object

## 5.8.7 RuleCondorcet

**class** whalrus.**RuleCondorcet**(**args*, *converter: whalrus.converters_ballot.converter_ballot.ConverterBallot = None*, *matrix_majority: whalrus.matrices.matrix.Matrix = None*, ***kwargs*)

Condorcet Rule.

**Parameters**

- **args** – Cf. parent class.
- **converter** (ConverterBallot) – Default: *ConverterBallotToOrder*.
- **matrix_majority** (Matrix) – The majority matrix. Default: *MatrixMajority*.
- **kwargs** – Cf. parent class.

### Examples

If there is a Condorcet winner, then it it the winner and all other candidates are tied. If there is no Condorcet winner, then all candidates are tied.

```
>>> RuleCondorcet(ballots=['a > b > c', 'b > a > c', 'c > a > b']).order_
[{'a'}, {'b', 'c'}]
>>> RuleCondorcet(ballots=['a > b > c', 'b > c > a', 'c > a > b']).order_
[{'a', 'b', 'c'}]
```

More precisely, and in all generality, a candidate is considered a *Condorcet winner* if all the non-diagonal coefficients on its raw of `matrix_majority_` are equal to 1. With the default setting of `matrix_majority = MatrixMajority()`, the Condorcet winner is necessarily unique when it exists, but that might not be the case with some more exotic settings:

```
>>> rule = RuleCondorcet(ballots=['a ~ b > c'], matrix_
↪majority=MatrixMajority(equal=1))
>>> rule.matrix_majority_.as_array_
array([[Fraction(1, 2), 1, 1],
       [1, Fraction(1, 2), 1],
       [0, 0, Fraction(1, 2)]], dtype=object)
>>> rule.order_
[{'a', 'b'}, {'c'}]
```

**`cotrailers_`**

“Cotrailers” of the election, i.e. the candidates that fare worst in the election. This is the last equivalence class in `order_`. For example, in `RuleScoreNum`, it is the candidates that are tied for the worst score.

**Type** *NiceSet*

**`cowinners_`**

Cowinners of the election, i.e. the candidates that fare best in the election.. This is the first equivalence class in `order_`. For example, in `RuleScoreNum`, it is the candidates that are tied for the best score.

**Type** *NiceSet*

**`matrix_majority_`**

The majority matrix (once computed with the given profile).

**Type** *Matrix*

**`n_candidates_`**

Number of candidates.

**Type** int

**`strict_order_`**

Result of the election as a strict order over the candidates. The first element is the winner, etc. This may use the tie-breaking rule.

**Type** list

**`trailer_`**

The “trailer” of the election. This is the last candidate in `strict_order_` and also the unfavorable choice of the tie-breaking rule in `cotrailers_`.

**Type** object

**`winner_`**

The winner of the election. This is the first candidate in `strict_order_` and also the choice of the tie-breaking rule in `cowinners_`.

**Type** object

### 5.8.8 RuleCoombs

**class** `whalrus.`**`RuleCoombs`**(**args, base_rule: whalrus.rules.rule.Rule = None, elimination: whalrus.eliminations.elimination.Elimination = None, **kwargs*)

Coombs’ rule.

**Parameters**

- **args** – Cf. parent class.
- **base_rule** (Rule) – Default: *RuleVeto*.
- **elimination** (Elimination) – Default: *EliminationLast* with k=1.
- **kwargs** – Cf. parent class.

### Examples

At each round, the candidate with the worst Veto score is eliminated.

```
>>> rule = RuleCoombs(['a > b > c', 'b > a > c', 'c > a > b'], weights=[2, 3, 4])
>>> rule.eliminations_[0].rule_.gross_scores_
{'a': 0, 'b': -4, 'c': -5}
>>> rule.eliminations_[1].rule_.gross_scores_
{'a': -3, 'b': -6}
>>> rule.eliminations_[2].rule_.gross_scores_
{'a': -9}
>>> rule.winner_
'a'
```

**cotrailers_**

“Cotrailers” of the election, i.e. the candidates that fare worst in the election. This is the last equivalence class in order_. For example, in *RuleScoreNum*, it is the candidates that are tied for the worst score.

**Type** *NiceSet*

**cowinners_**

Cowinners of the election, i.e. the candidates that fare best in the election.. This is the first equivalence class in order_. For example, in *RuleScoreNum*, it is the candidates that are tied for the best score.

**Type** *NiceSet*

**eliminations_**

The elimination rounds. A list of *Elimination* objects. The first one corresponds to the first round, etc.

**Type** list

**n_candidates_**

Number of candidates.

**Type** int

**strict_order_**

Result of the election as a strict order over the candidates. The first element is the winner, etc. This may use the tie-breaking rule.

**Type** list

**trailer_**

The “trailer” of the election. This is the last candidate in *strict_order_* and also the unfavorable choice of the tie-breaking rule in *cotrailers_*.

**Type** object

**winner_**

The winner of the election. This is the first candidate in *strict_order_* and also the choice of the tie-breaking rule in *cowinners_*.

**Type** object

### 5.8.9 RuleCopeland

**class** whalrus.**RuleCopeland**(**args*, *converter: whalrus.converters_ballot.converter_ballot.ConverterBallot = None*, *matrix: whalrus.matrices.matrix.Matrix = None*, ***kwargs*)

Copeland's rule.

**Parameters**

- **args** – Cf. parent class.
- **converter** (ConverterBallot) – Default: *ConverterBallotToOrder*.
- **matrix** (Matrix) – Default: *MatrixMajority*.
- **kwargs** – Cf. parent class.

#### Examples

The score of a candidate is the number of victories in the majority matrix.

```
>>> rule = RuleCopeland(ballots=['a > b > c', 'b > a > c', 'c > a > b'])
>>> rule.matrix_.as_array_
array([[Fraction(1, 2), 1, 1],
       [0, Fraction(1, 2), 1],
       [0, 0, Fraction(1, 2)]], dtype=object)
>>> rule.scores_
{'a': 2, 'b': 1, 'c': 0}
```

**average_score_**

The average score.

**Type** Number

**average_score_as_float_**

The average score as a float. It is the same as *average_score_*, but converted to a float.

**Type** float

**best_score_as_float_**

The best score as a float. It is the same as *RuleScore.best_score_*, but converted to a float.

**Type** float

**compare_scores**(*one: numbers.Number*, *another: numbers.Number*) → int

Compare two scores.

**Parameters**

- **one** (*object*) – A score.
- **another** (*object*) – A score.

**Returns** 0 if they are equal, a positive number if one is greater than another, a negative number otherwise.

**Return type** int

**cotrailers_**

"Cotrailers". The set of candidates with the worst score.

**Type** *NiceSet*

**cowinners_**

Cowinners. The set of candidates with the best score.

**Type** *NiceSet*

**matrix_**

The matrix (once computed with the given profile).

**Type** *Matrix*

**matrix_majority_**

The majority matrix. This is an alias for *matrix_*.

### Examples

```
>>> rule = RuleCopeland(ballots=['a > b > c', 'b > a > c', 'c > a > b'])
>>> rule.matrix_majority_.as_array_
array([[Fraction(1, 2), 1, 1],
       [0, Fraction(1, 2), 1],
       [0, 0, Fraction(1, 2)]], dtype=object)
```

**Type** *Matrix*

**n_candidates_**

Number of candidates.

**Type** int

**scores_as_floats_**

Scores as floats. It is the same as `scores_`, but converted to floats.

**Type** *NiceDict*

**strict_order_**

Result of the election as a strict order over the candidates. The first element is the winner, etc. This may use the tie-breaking rule.

**Type** list

**trailer_**

The "trailer" of the election. This is the last candidate in *strict_order_* and also the unfavorable choice of the tie-breaking rule in *cotrailers_*.

**Type** object

**winner_**

The winner of the election. This is the first candidate in *strict_order_* and also the choice of the tie-breaking rule in *cowinners_*.

**Type** object

**worst_score_as_float_**

The worst score as a float. It is the same as *RuleScore.worst_score_*, but converted to a float.

**Type** float

## 5.8.10 RuleIRV

**class** whalrus.**RuleIRV**(**args, base_rule: whalrus.rules.rule.Rule = None, elimination: whalrus.eliminations.elimination.Elimination = None, **kwargs*)

Instant-Runoff Voting, also known as Alternative Vote, Single Transferable Vote, etc.

**Parameters**

- **args** – Cf. parent class.
- **base_rule** (Rule) – Default: *RulePlurality*.
- **elimination** (Elimination) – Default: *EliminationLast* with k=1.
- **kwargs** – Cf. parent class.

### Examples

At each round, the candidate with the worst Plurality score is eliminated.

```
>>> rule = RuleIRV(['a > b > c', 'b > a > c', 'c > a > b'], weights=[2, 3, 4])
>>> rule.eliminations_[0].rule_.gross_scores_
{'a': 2, 'b': 3, 'c': 4}
>>> rule.eliminations_[1].rule_.gross_scores_
{'b': 5, 'c': 4}
>>> rule.eliminations_[2].rule_.gross_scores_
{'b': 9}
>>> rule.winner_
'b'
```

An example using the tie-break:

```
>>> rule = RuleIRV(['a > c > b', 'b > a > c', 'c > a > b'], weights=[1, 2, 1],
...                tie_break=Priority.ASCENDING)
>>> rule.eliminations_[0].rule_.gross_scores_
{'a': 1, 'b': 2, 'c': 1}
>>> rule.eliminations_[1].rule_.gross_scores_
{'a': 2, 'b': 2}
>>> rule.eliminations_[2].rule_.gross_scores_
{'a': 4}
>>> rule.winner_
'a'
```

**cotrailers_**

“Cotrailers” of the election, i.e. the candidates that fare worst in the election. This is the last equivalence class in order_. For example, in *RuleScoreNum*, it is the candidates that are tied for the worst score.

**Type** *NiceSet*

**cowinners_**

Cowinners of the election, i.e. the candidates that fare best in the election.. This is the first equivalence class in order_. For example, in *RuleScoreNum*, it is the candidates that are tied for the best score.

**Type** *NiceSet*

**eliminations_**

The elimination rounds. A list of *Elimination* objects. The first one corresponds to the first round, etc.

**Type** list

**n_candidates_**

Number of candidates.

**Type** int

**strict_order_**

Result of the election as a strict order over the candidates. The first element is the winner, etc. This may use the tie-breaking rule.

**Type** list

**trailer_**

The "trailer" of the election. This is the last candidate in *strict_order_* and also the unfavorable choice of the tie-breaking rule in *cotrailers_*.

**Type** object

**winner_**

The winner of the election. This is the first candidate in *strict_order_* and also the choice of the tie-breaking rule in *cowinners_*.

**Type** object

### 5.8.11 RuleKApproval

**class** whalrus.**RuleKApproval**(**args*, *k: int = 1*, ***kwargs*)

K-Approval

The k top candidates in a ballot receive 1 point, and the other candidates receive 0 point.

**Parameters**

- **args** – Cf. parent class.
- **k** (*int*) – The number of approved candidates.
- **kwargs** – Cf. parent class.

#### Examples

```
>>> RuleKApproval(['a > b > c', 'b > c > a'], k=2).gross_scores_
{'a': 1, 'b': 2, 'c': 1}
```

**average_score_**

The average score.

**Type** Number

**average_score_as_float_**

The average score as a float. It is the same as *average_score_*, but converted to a float.

**Type** float

**best_score_as_float_**

The best score as a float. It is the same as *RuleScore.best_score_*, but converted to a float.

**Type** float

**compare_scores**(*one: numbers.Number*, *another: numbers.Number*) → int

Compare two scores.

**Parameters**

- **one** (*object*) – A score.
- **another** (*object*) – A score.

**Returns** 0 if they are equal, a positive number if one is greater than another, a negative number otherwise.

**Return type** int

**cotrailers_**

“Cotrailers”. The set of candidates with the worst score.

**Type** *NiceSet*

**cowinners_**

Cowinners. The set of candidates with the best score.

**Type** *NiceSet*

**gross_scores_**

The gross scores of the candidates. For each candidate, this dictionary gives the sum of its scores, multiplied by the weights of the corresponding voters. This is the numerator in the candidate’s average score.

**Type** *NiceDict*

**gross_scores_as_floats_**

Gross scores as floats. It is the same as `gross_scores_`, but converted to floats.

**Type** *NiceDict*

**n_candidates_**

Number of candidates.

**Type** int

**scores_as_floats_**

Scores as floats. It is the same as `scores_`, but converted to floats.

**Type** *NiceDict*

**strict_order_**

Result of the election as a strict order over the candidates. The first element is the winner, etc. This may use the tie-breaking rule.

**Type** list

**trailer_**

The “trailer” of the election. This is the last candidate in `strict_order_` and also the unfavorable choice of the tie-breaking rule in `cotrailers_`.

**Type** object

**weights_**

The weights used for the candidates. For each candidate, this dictionary gives the total weight for this candidate, i.e. the total weight of all voters who assign a score to this candidate. This is the denominator in the candidate’s average score.

**Type** *NiceDict*

**weights_as_floats_**

Weights as floats. It is the same as `weights_`, but converted to floats.

**Type** *NiceDict*

**winner_**

The winner of the election. This is the first candidate in `strict_order_` and also the choice of the tie-breaking rule in `cowinners_`.

**Type** object

**worst_score_as_float_**

The worst score as a float. It is the same as `RuleScore.worst_score_`, but converted to a float.

**Type** float

### 5.8.12 RuleKimRoush

**class** whalrus.**RuleKimRoush**(**args, base_rule: whalrus.rules.rule.Rule = None, elimination: whalrus.eliminations.elimination.Elimination = None, **kwargs*)

Kim-Roush rule.

At each round, all candidates whose Veto score is lower than the average Veto score are eliminated.

**Parameters**

- **args** – Cf. parent class.
- **base_rule** (Rule) – Default: *RuleVeto*.
- **elimination** (Elimination) – Default: *EliminationBelowAverage*.
- **kwargs** – Cf. parent class.

#### Examples

```
>>> rule = RuleKimRoush(['a > b > c > d', 'a > b > d > c'])
>>> rule.eliminations_[0].rule_.gross_scores_
{'a': 0, 'b': 0, 'c': -1, 'd': -1}
>>> rule.eliminations_[1].rule_.gross_scores_
{'a': 0, 'b': -2}
>>> rule.eliminations_[2].rule_.gross_scores_
{'a': -2}
>>> rule.winner_
'a'
```

**cotrailers_**

“Cotrailers” of the election, i.e. the candidates that fare worst in the election. This is the last equivalence class in order_. For example, in *RuleScoreNum*, it is the candidates that are tied for the worst score.

**Type** *NiceSet*

**cowinners_**

Cowinners of the election, i.e. the candidates that fare best in the election.. This is the first equivalence class in order_. For example, in *RuleScoreNum*, it is the candidates that are tied for the best score.

**Type** *NiceSet*

**eliminations_**

The elimination rounds. A list of *Elimination* objects. The first one corresponds to the first round, etc.

**Type** list

**n_candidates_**

Number of candidates.

**Type** int

**strict_order_**

Result of the election as a strict order over the candidates. The first element is the winner, etc. This may use the tie-breaking rule.

**Type** list

**trailer_**

The “trailer” of the election. This is the last candidate in *strict_order_* and also the unfavorable choice of the tie-breaking rule in *cotrailers_*.

**Type** object

**winner_**

The winner of the election. This is the first candidate in *strict_order_* and also the choice of the tie-breaking rule in *cowinners_*.

**Type** object

## 5.8.13 RuleMajorityJudgment

**class** whalrus.**RuleMajorityJudgment**(**args, converter: whalrus.converters_ballot.converter_ballot.ConverterBallot = None, scorer: whalrus.scorers.scorer.Scorer = None, scale: whalrus.scales.scale.Scale = None, default_median: object = None, **kwargs*)

Majority Judgment.

**Parameters**

- **args** – Cf. parent class.
- **converter** (ConverterBallot) – Default: *ConverterBallotToLevels*, with `scale=scorer.scale`.
- **scorer** (Scorer) – Default: *ScorerLevels*. Alternatively, you may provide an argument `scale`. In that case, the scorer will be `ScorerLevels(scale)`.
- **default_median** (*object*) – The median level that a candidate has when it receives absolutely no evaluation whatsoever.
- **kwargs** – Cf. parent class.

### Examples

```
>>> rule = RuleMajorityJudgment([{'a': 1, 'b': 1}, {'a': .5, 'b': .6},
...                              {'a': .5, 'b': .4}, {'a': .3, 'b': .2}])
>>> rule.scores_as_floats_
{'a': (0.5, -0.25, 0.25), 'b': (0.4, 0.5, -0.25)}
>>> rule.winner_
'a'
```

For each candidate, its median evaluation *m* is computed. When a candidate has two medians (like candidate *b* in the above example, with .4 and .6), the lower value is considered. Let *p* (resp. *q*) denote the proportion of the voters who evaluate the candidate better (resp. worse) than its median. The score of the candidate is the tuple *(m, p, -q)* if *p* > *q*, and *(m, -q, p)* otherwise. Scores are compared lexicographically.

For Majority Judgment, verbal evaluation are generally used. The following example is actually the same as above, but with verbal evaluations instead of grades:

```
>>> rule = RuleMajorityJudgment([
...     {'a': 'Excellent', 'b': 'Excellent'}, {'a': 'Good', 'b': 'Very Good'},
...     {'a': 'Good', 'b': 'Acceptable'}, {'a': 'Poor', 'b': 'To Reject'}
... ], scale=ScaleFromList(['To Reject', 'Poor', 'Acceptable', 'Good', 'Very Good
↪', 'Excellent']))
>>> rule.scores_as_floats_
{'a': ('Good', -0.25, 0.25), 'b': ('Acceptable', 0.5, -0.25)}
>>> rule.winner_
'a'
```

By changing the `scorer`, you may define a very different rule. The following one rewards the candidate with best median Borda score (with secondary criteria that are similar to Majority Judgment, i.e. the proportions of voters who give a candidate more / less than its median Borda score):

```
>>> from whalrus.scorers.scorer_borda import ScorerBorda
>>> from whalrus.converters_ballot.converter_ballot_to_order import␣
↪ConverterBallotToOrder
>>> rule = RuleMajorityJudgment(scorer=ScorerBorda(),␣
↪converter=ConverterBallotToOrder())
>>> rule(['a > b ~ c > d', 'c > a > b > d']).scores_as_floats_
{'a': (2.0, 0.5, 0.0), 'b': (1.0, 0.5, 0.0), 'c': (1.5, 0.5, 0.0), 'd': (0.0, 0.0,
↪ 0.0)}
>>> rule.winner_
'a'
```

**`best_score_`**
The best score.

**Type** object

**`compare_scores`** (*one: tuple*, *another: tuple*) → int
Compare two scores.

**Parameters**

- **one** (*`object`*) – A score.
- **another** (*`object`*) – A score.

**Returns** 0 if they are equal, a positive number if `one` is greater than `another`, a negative number otherwise.

**Return type** int

**`cotrailers_`**
"Cotrailers". The set of candidates with the worst score.

**Type** *NiceSet*

**`cowinners_`**
Cowinners. The set of candidates with the best score.

**Type** *NiceSet*

**`n_candidates_`**
Number of candidates.

**Type** int

**`order_`**
Result of the election as a (weak) order over the candidates. It is a list of `NiceSet`. The first set contains the candidates that have the best score, the second set contains those with the second best score, etc.

**Type** list

**`scores_`**
The scores. A `NiceDict` of triples.

**Type** *NiceDict*

**`scores_as_floats_`**
Scores as floats. It is the same as *`scores_`*, but converted to floats.

**Type** *NiceDict*

**strict_order_**

Result of the election as a strict order over the candidates. The first element is the winner, etc. This may use the tie-breaking rule.

**Type** list

**trailer_**

The "trailer" of the election. This is the last candidate in *strict_order_* and also the unfavorable choice of the tie-breaking rule in *cotrailers_*.

**Type** object

**winner_**

The winner of the election. This is the first candidate in *strict_order_* and also the choice of the tie-breaking rule in *cowinners_*.

**Type** object

**worst_score_**

The worst score.

**Type** object

## 5.8.14 RuleMaximin

**class** whalrus.**RuleMaximin**(**args*, *converter: whalrus.converters_ballot.converter_ballot.ConverterBallot = None*, *matrix_weighted_majority: whalrus.matrices.matrix.Matrix = None*, ***kwargs*)

Maximin rule. Also known as Simpson-Kramer rule.

The score of a candidate is the minimal non-diagonal coefficient on its raw of the matrix.

**Parameters**

- **args** – Cf. parent class.
- **converter** (ConverterBallot) – Default: *ConverterBallotToOrder*.
- **matrix_weighted_majority** (Matrix) – Default: *MatrixWeightedMajority*.
- **kwargs** – Cf. parent class.

### Examples

```
>>> rule = RuleMaximin(ballots=['a > b > c', 'b > c > a', 'c > a > b'],␣
↪weights=[4, 3, 3])
>>> rule.matrix_weighted_majority_.as_array_of_floats_
array([[0. , 0.7, 0.4],
       [0.3, 0. , 0.7],
       [0.6, 0.3, 0. ]])
>>> rule.scores_as_floats_
{'a': 0.4, 'b': 0.3, 'c': 0.3}
>>> rule.winner_
'a'
```

**average_score_**

The average score.

**Type** Number

**average_score_as_float_**

The average score as a float. It is the same as *average_score_*, but converted to a float.

**Type** float

**best_score_as_float_**

The best score as a float. It is the same as *RuleScore.best_score_*, but converted to a float.

**Type** float

**compare_scores** (*one: numbers.Number*, *another: numbers.Number*) → int

Compare two scores.

**Parameters**

- **one** (*object*) – A score.
- **another** (*object*) – A score.

**Returns** 0 if they are equal, a positive number if `one` is greater than `another`, a negative number otherwise.

**Return type** int

**cotrailers_**

“Cotrailers”. The set of candidates with the worst score.

**Type** *NiceSet*

**cowinners_**

Cowinners. The set of candidates with the best score.

**Type** *NiceSet*

**matrix_weighted_majority_**

The weighted majority matrix (once computed with the given profile).

**Type** *Matrix*

**n_candidates_**

Number of candidates.

**Type** int

**scores_as_floats_**

Scores as floats. It is the same as `scores_`, but converted to floats.

**Type** *NiceDict*

**strict_order_**

Result of the election as a strict order over the candidates. The first element is the winner, etc. This may use the tie-breaking rule.

**Type** list

**trailer_**

The “trailer” of the election. This is the last candidate in *strict_order_* and also the unfavorable choice of the tie-breaking rule in *cotrailers_*.

**Type** object

**winner_**

The winner of the election. This is the first candidate in *strict_order_* and also the choice of the tie-breaking rule in *cowinners_*.

**Type** object

**worst_score_as_float_**

The worst score as a float. It is the same as *RuleScore.worst_score_*, but converted to a float.

**Type** float

## 5.8.15 RuleNanson

**class** whalrus.**RuleNanson**(**args, base_rule: whalrus.rules.rule.Rule = None, elimination: whalrus.eliminations.elimination.Elimination = None, **kwargs*)

Nanson's rule.

At each round, all candidates whose Borda score is lower than the average Borda score are eliminated.

**Parameters**

- **args** – Cf. parent class.
- **base_rule** (Rule) – Default: *RuleBorda*.
- **elimination** (Elimination) – Default: *EliminationBelowAverage*.
- **kwargs** – Cf. parent class.

### Examples

```
>>> rule = RuleNanson(['a > b > c > d', 'a > b > d > c'])
>>> rule.eliminations_[0].rule_.gross_scores_
{'a': 6, 'b': 4, 'c': 1, 'd': 1}
>>> rule.eliminations_[1].rule_.gross_scores_
{'a': 2, 'b': 0}
>>> rule.eliminations_[2].rule_.gross_scores_
{'a': 0}
>>> rule.winner_
'a'
```

**cotrailers_**

"Cotrailers" of the election, i.e. the candidates that fare worst in the election. This is the last equivalence class in order_. For example, in *RuleScoreNum*, it is the candidates that are tied for the worst score.

**Type** *NiceSet*

**cowinners_**

Cowinners of the election, i.e. the candidates that fare best in the election.. This is the first equivalence class in order_. For example, in *RuleScoreNum*, it is the candidates that are tied for the best score.

**Type** *NiceSet*

**eliminations_**

The elimination rounds. A list of *Elimination* objects. The first one corresponds to the first round, etc.

**Type** list

**n_candidates_**

Number of candidates.

**Type** int

**strict_order_**

Result of the election as a strict order over the candidates. The first element is the winner, etc. This may use the tie-breaking rule.

**Type** list

**trailer_**

The "trailer" of the election. This is the last candidate in *strict_order_* and also the unfavorable choice of the tie-breaking rule in *cotrailers_*.

**Type** object

**winner_**

The winner of the election. This is the first candidate in *strict_order_* and also the choice of the tie-breaking rule in *cowinners_*.

**Type** object

## 5.8.16 RulePlurality

**class** whalrus.**RulePlurality**(**args*, *converter: whalrus.converters_ballot.converter_ballot.ConverterBallot = None*, *scorer: whalrus.scorers.scorer.Scorer = None*, ***kwargs*)

The plurality rule.

**Parameters**

- **args** – Cf. parent class.
- **converter** (ConverterBallot) – Default: *ConverterBallotToPlurality*.
- **scorer** (Scorer) – Default: *ScorerPlurality*.
- **kwargs** – Cf. parent class.

### Examples

In the most general syntax, firstly, you define the rule:

```
>>> plurality = RulePlurality(tie_break=Priority.ASCENDING)
```

Secondly, you use it as a callable to load a particular election (profile, candidates):

```
>>> plurality(ballots=['a', 'b', 'c'], weights=[2, 2, 1], voters=['Alice', 'Bob',
↪'Cate'],
...           candidates={'a', 'b', 'c', 'd'})  # doctest:+ELLIPSIS
<... object at ...>
```

Finally, you can access the computed variables:

```
>>> plurality.gross_scores_
{'a': 2, 'b': 2, 'c': 1, 'd': 0}
>>> plurality.winner_
'a'
```

Later, if you wish, you can load another profile with the same voting rule, and so on.

Optionally, you can specify an election (profile and candidates) as soon as the *Rule* object is initialized. This allows for one-liners such as:

```
>>> RulePlurality(['a', 'a', 'b', 'c']).winner_
'a'
```

Cf. *Rule* for more information about the arguments.

**average_score_**

The average score.

**Type** Number

**average_score_as_float_**

The average score as a float. It is the same as *average_score_*, but converted to a float.

**Type** float

**best_score_as_float_**

The best score as a float. It is the same as *RuleScore.best_score_*, but converted to a float.

**Type** float

**compare_scores**(*one: numbers.Number*, *another: numbers.Number*) → int

Compare two scores.

**Parameters**

- **one** (*object*) – A score.
- **another** (*object*) – A score.

**Returns** 0 if they are equal, a positive number if `one` is greater than `another`, a negative number otherwise.

**Return type** int

**cotrailers_**

"Cotrailers". The set of candidates with the worst score.

**Type** *NiceSet*

**cowinners_**

Cowinners. The set of candidates with the best score.

**Type** *NiceSet*

**gross_scores_as_floats_**

Gross scores as floats. It is the same as `gross_scores_`, but converted to floats.

**Type** *NiceDict*

**n_candidates_**

Number of candidates.

**Type** int

**scores_as_floats_**

Scores as floats. It is the same as `scores_`, but converted to floats.

**Type** *NiceDict*

**strict_order_**

Result of the election as a strict order over the candidates. The first element is the winner, etc. This may use the tie-breaking rule.

**Type** list

**trailer_**

The "trailer" of the election. This is the last candidate in *strict_order_* and also the unfavorable choice of the tie-breaking rule in *cotrailers_*.

**Type** object

**weights_as_floats_**

Weights as floats. It is the same as `weights_`, but converted to floats.

**Type** *NiceDict*

**winner_**

The winner of the election. This is the first candidate in *strict_order_* and also the choice of the tie-breaking rule in *cowinners_*.

**Type** object

**worst_score_as_float_**

The worst score as a float. It is the same as *RuleScore.worst_score_*, but converted to a float.

**Type** float

## 5.8.17 RuleRangeVoting

**class** whalrus.**RuleRangeVoting**(**args, converter: whalrus.converters_ballot.converter_ballot.ConverterBallot = None, scorer: whalrus.scorers.scorer.Scorer = None, **kwargs*)

Range voting.

**Parameters**

- **args** – Cf. parent class.
- **converter** (`ConverterBallot`) – Default: *ConverterBallotToGrades*.
- **scorer** (`Scorer`) – Default: *ScorerLevels*.
- **kwargs** – Cf. parent class.

### Examples

Typical usage:

```
>>> RuleRangeVoting([{'a': 1, 'b': .8, 'c': .2}, {'a': 0, 'b': .6, 'c': 1}]).
↪scores_
{'a': Fraction(1, 2), 'b': Fraction(7, 10), 'c': Fraction(3, 5)}
>>> RuleRangeVoting([{'a': 10, 'b': 8, 'c': 2}, {'a': 0, 'b': 6, 'c': 10}]).
↪scores_
{'a': 5, 'b': 7, 'c': 6}
```

The following examples use the ballot converter:

```
>>> RuleRangeVoting(['a > b > c']).profile_converted_[0].as_dict
{'a': 1, 'b': Fraction(1, 2), 'c': 0}
>>> RuleRangeVoting(
...     ['a > b > c'], converter=ConverterBallotToGrades(scale=ScaleRange(0, 10))
... ).profile_converted_[0].as_dict
{'a': 10, 'b': 5, 'c': 0}
```

To examine the effect of the options, let us examine:

```
>>> b1 = BallotLevels({'a': 8, 'b': 10}, candidates={'a', 'b'})
>>> b2 = BallotLevels({'a': 6, 'c': 10}, candidates={'a', 'b', 'c'})
```

In ballot b1, candidate *c* is absent, which means that the candidate was not even available when the voter cast her ballot. In ballot b2, candidate *b* is ungraded: it was available, but the voter decided not to give it a grade. By the way, we will also introduce a candidate *d* who receives no evaluation at all. Here are several possible settings for the voting rule, along with their consequences:

```
>>> RuleRangeVoting([b1, b2], candidates={'a', 'b', 'c', 'd'}).scores_
{'a': 7, 'b': 10, 'c': 10, 'd': 0}
>>> RuleRangeVoting([b1, b2], candidates={'a', 'b', 'c', 'd'}, default_average=5).
↪scores_
{'a': 7, 'b': 10, 'c': 10, 'd': 5}
>>> RuleRangeVoting([b1, b2], candidates={'a', 'b', 'c', 'd'},
...     scorer=ScorerLevels(level_ungraded=0)).scores_
{'a': 7, 'b': 5, 'c': 10, 'd': 0}
>>> RuleRangeVoting([b1, b2], candidates={'a', 'b', 'c', 'd'},
...     scorer=ScorerLevels(level_ungraded=0, level_absent=0)).scores_
{'a': 7, 'b': 5, 'c': 5, 'd': 0}
```

For more information, cf. `ScorerLevels`.

**average_score_**

The average score.

**Type** Number

**average_score_as_float_**

The average score as a float. It is the same as `average_score_`, but converted to a float.

**Type** float

**best_score_as_float_**

The best score as a float. It is the same as `RuleScore.best_score_`, but converted to a float.

**Type** float

**compare_scores**(*one: numbers.Number*, *another: numbers.Number*) → int

Compare two scores.

**Parameters**

- **one** (*object*) – A score.
- **another** (*object*) – A score.

**Returns** 0 if they are equal, a positive number if `one` is greater than `another`, a negative number otherwise.

**Return type** int

**cotrailers_**

"Cotrailers". The set of candidates with the worst score.

**Type** *NiceSet*

**cowinners_**

Cowinners. The set of candidates with the best score.

**Type** *NiceSet*

**gross_scores_**

The gross scores of the candidates. For each candidate, this dictionary gives the sum of its scores, multiplied by the weights of the corresponding voters. This is the numerator in the candidate's average score.

**Type** *NiceDict*

**gross_scores_as_floats_**
Gross scores as floats. It is the same as *gross_scores_*, but converted to floats.

**Type** *NiceDict*

**n_candidates_**
Number of candidates.

**Type** int

**scores_as_floats_**
Scores as floats. It is the same as `scores_`, but converted to floats.

**Type** *NiceDict*

**strict_order_**
Result of the election as a strict order over the candidates. The first element is the winner, etc. This may use the tie-breaking rule.

**Type** list

**trailer_**
The "trailer" of the election. This is the last candidate in *strict_order_* and also the unfavorable choice of the tie-breaking rule in *cotrailers_*.

**Type** object

**weights_**
The weights used for the candidates. For each candidate, this dictionary gives the total weight for this candidate, i.e. the total weight of all voters who assign a score to this candidate. This is the denominator in the candidate's average score.

**Type** *NiceDict*

**weights_as_floats_**
Weights as floats. It is the same as *weights_*, but converted to floats.

**Type** *NiceDict*

**winner_**
The winner of the election. This is the first candidate in *strict_order_* and also the choice of the tie-breaking rule in *cowinners_*.

**Type** object

**worst_score_as_float_**
The worst score as a float. It is the same as *RuleScore.worst_score_*, but converted to a float.

**Type** float

### 5.8.18 RuleRankedPairs

**class** whalrus.**RuleRankedPairs**(**args*, *converter: whalrus.converters_ballot.converter_ballot.ConverterBallot = None*, *matrix: whalrus.matrices.matrix.Matrix = None*, ***kwargs*)

Ranked Pairs rule.

The score of a candidate is the number of victories in the ranked pairs matrix.

**Parameters**

- **args** – Cf. parent class.
- **converter** (`ConverterBallot`) – Default: *ConverterBallotToOrder*.

- **matrix** (Matrix) – Default: `MatrixRankedPairs(tie_break=tie_break)`.
- **kwargs** – Cf. parent class.

### Examples

```
>>> rule = RuleRankedPairs(['a > b > c', 'b > c > a', 'c > a > b'], weights=[4, 3,
↪ 2])
>>> rule.matrix_.as_array_
array([[0, 1, 1],
       [0, 0, 1],
       [0, 0, 0]], dtype=object)
>>> rule.scores_
{'a': 2, 'b': 1, 'c': 0}
```

**average_score_**

The average score.

**Type** Number

**average_score_as_float_**

The average score as a float. It is the same as *average_score_*, but converted to a float.

**Type** float

**best_score_as_float_**

The best score as a float. It is the same as *RuleScore.best_score_*, but converted to a float.

**Type** float

**compare_scores** (*one: numbers.Number*, *another: numbers.Number*) → int

Compare two scores.

**Parameters**

- **one** (*object*) – A score.
- **another** (*object*) – A score.

**Returns** 0 if they are equal, a positive number if `one` is greater than `another`, a negative number otherwise.

**Return type** int

**cotrailers_**

“Cotrailers”. The set of candidates with the worst score.

**Type** *NiceSet*

**cowinners_**

Cowinners. The set of candidates with the best score.

**Type** *NiceSet*

**matrix_**

The matrix (once computed with the given profile).

**Type** *Matrix*

**matrix_ranked_pairs_**

The ranked pairs matrix. Alias for *matrix_*.

### Examples

```
>>> rule = RuleRankedPairs(['a > b > c', 'b > c > a', 'c > a > b'],␣
↪weights=[4, 3, 2])
>>> rule.matrix_ranked_pairs_.as_array_
array([[0, 1, 1],
       [0, 0, 1],
       [0, 0, 0]], dtype=object)
```

**Type** *Matrix*

**n_candidates_**

Number of candidates.

**Type** int

**scores_as_floats_**

Scores as floats. It is the same as `scores_`, but converted to floats.

**Type** *NiceDict*

**strict_order_**

Result of the election as a strict order over the candidates. The first element is the winner, etc. This may use the tie-breaking rule.

**Type** list

**trailer_**

The "trailer" of the election. This is the last candidate in `strict_order_` and also the unfavorable choice of the tie-breaking rule in `cotrailers_`.

**Type** object

**winner_**

The winner of the election. This is the first candidate in `strict_order_` and also the choice of the tie-breaking rule in `cowinners_`.

**Type** object

**worst_score_as_float_**

The worst score as a float. It is the same as `RuleScore.worst_score_`, but converted to a float.

**Type** float

## 5.8.19 RuleSchulze

**class** whalrus.**RuleSchulze**(**args, converter: whalrus.converters_ballot.converter_ballot.ConverterBallot = None, matrix_schulze: whalrus.matrices.matrix.Matrix = None, **kwargs*)

Schulze's Rule.

A candidate is a Schulze winner if it has no defeat in the Schulze matrix.

**Parameters**

- **args** – Cf. parent class.
- **converter** (`ConverterBallot`) – Default: `ConverterBallotToOrder`.
- **matrix_schulze** (`Matrix`) – The Schulze matrix. Default: `MatrixSchulze`.

- **kwargs** – Cf. parent class.

### Examples

```
>>> rule = RuleSchulze(['a > b > c', 'b > c > a', 'c > a > b'], weights=[4, 3, 2])
>>> rule.matrix_schulze_.as_array_
array([[0, Fraction(2, 3), Fraction(2, 3)],
       [Fraction(5, 9), 0, Fraction(7, 9)],
       [Fraction(5, 9), Fraction(5, 9), 0]], dtype=object)
>>> rule.winner_
'a'
```

**cotrailers_**

“Cotrailers” of the election, i.e. the candidates that fare worst in the election. This is the last equivalence class in order_. For example, in *RuleScoreNum*, it is the candidates that are tied for the worst score.

**Type** *NiceSet*

**cowinners_**

Cowinners of the election, i.e. the candidates that fare best in the election.. This is the first equivalence class in order_. For example, in *RuleScoreNum*, it is the candidates that are tied for the best score.

**Type** *NiceSet*

**matrix_schulze_**

The Schulze matrix (once computed with the given profile).

**Type** *Matrix*

**n_candidates_**

Number of candidates.

**Type** int

**strict_order_**

Result of the election as a strict order over the candidates. The first element is the winner, etc. This may use the tie-breaking rule.

**Type** list

**trailer_**

The “trailer” of the election. This is the last candidate in *strict_order_* and also the unfavorable choice of the tie-breaking rule in *cotrailers_*.

**Type** object

**winner_**

The winner of the election. This is the first candidate in *strict_order_* and also the choice of the tie-breaking rule in *cowinners_*.

**Type** object

## 5.8.20 RuleSimplifiedDodgson

**class** whalrus.**RuleSimplifiedDodgson**(**args, converter: whalrus.converters_ballot.converter_ballot.ConverterBallot = None, matrix_weighted_majority: whalrus.matrices.matrix.Matrix = None, **kwargs*)

Simplified Dodgson rule.

The score of a candidate is the sum of the negative non-diagonal coefficient on its raw of *matrix_weighted_majority_*.

**Parameters**

- **args** – Cf. parent class.
- **converter** (ConverterBallot) – Default: *ConverterBallotToOrder*.
- **matrix_weighted_majority** (Matrix) – Default: *MatrixWeightedMajority* with antisymmetric=True.
- **kwargs** – Cf. parent class.

### Examples

```
>>> rule = RuleSimplifiedDodgson(ballots=['a > b > c', 'b > a > c', 'c > a > b'],
...                              weights=[3, 3, 2])
>>> rule.matrix_weighted_majority_.as_array_
array([[0, Fraction(1, 4), Fraction(1, 2)],
       [Fraction(-1, 4), 0, Fraction(1, 2)],
       [Fraction(-1, 2), Fraction(-1, 2), 0]], dtype=object)
>>> rule.scores_
{'a': 0, 'b': Fraction(-1, 4), 'c': -1}
>>> rule.winner_
'a'
```

**average_score_**

The average score.

**Type** Number

**average_score_as_float_**

The average score as a float. It is the same as *average_score_*, but converted to a float.

**Type** float

**best_score_as_float_**

The best score as a float. It is the same as *RuleScore.best_score_*, but converted to a float.

**Type** float

**compare_scores**(*one: numbers.Number*, *another: numbers.Number*) → int

Compare two scores.

**Parameters**

- **one** (*object*) – A score.
- **another** (*object*) – A score.

**Returns** 0 if they are equal, a positive number if one is greater than another, a negative number otherwise.

**Return type** int

**cotrailers_**

"Cotrailers". The set of candidates with the worst score.

**Type** *NiceSet*

**cowinners_**

Cowinners. The set of candidates with the best score.

**Type** *NiceSet*

**matrix_weighted_majority_**
The weighted majority matrix (once computed with the given profile).

**Type** *Matrix*

**n_candidates_**
Number of candidates.

**Type** int

**scores_as_floats_**
Scores as floats. It is the same as `scores_`, but converted to floats.

**Type** *NiceDict*

**strict_order_**
Result of the election as a strict order over the candidates. The first element is the winner, etc. This may use the tie-breaking rule.

**Type** list

**trailer_**
The "trailer" of the election. This is the last candidate in `strict_order_` and also the unfavorable choice of the tie-breaking rule in `cotrailers_`.

**Type** object

**winner_**
The winner of the election. This is the first candidate in `strict_order_` and also the choice of the tie-breaking rule in `cowinners_`.

**Type** object

**worst_score_as_float_**
The worst score as a float. It is the same as `RuleScore.worst_score_`, but converted to a float.

**Type** float

### 5.8.21 RuleTwoRound

**class** whalrus.**RuleTwoRound**(**args, rule1: whalrus.rules.rule.Rule = None, rule2: whalrus.rules.rule.Rule = None, elimination: whalrus.eliminations.elimination.Elimination = None, **kwargs*)
The two-round system.

**Parameters**

- **args** – Cf. parent class.
- **rule1** – The first rule. Default: `RulePlurality`.
- **rule2** – The second rule. Default: `RulePlurality`.
- **elimination** (`Elimination`) – The elimination algorithm used during the first round. Default: `EliminationLast` with `k=-2`, which only keeps the 2 best candidates.
- **kwargs** – Cf. parent class.

### Examples

With its default settings, this class implements the classic two-round system, using plurality at both rounds:

```
>>> rule = RuleTwoRound(['a > b > c > d > e', 'b > a > c > d > e', 'c > a > b > d␣
↪> e'],
...                     weights=[2, 2, 1])
>>> rule.first_round_.rule_.gross_scores_
{'a': 2, 'b': 2, 'c': 1, 'd': 0, 'e': 0}
>>> rule.second_round_.gross_scores_
{'a': 3, 'b': 2}
```

Using the options, some more exotic two-round systems can be defined, such as changing the rule of a round:

```
>>> rule = RuleTwoRound(['a > b > c > d > e', 'b > a > c > d > e', 'c > a > b > d␣
↪> e'],
...                     weights=[2, 2, 1], rule1=RuleBorda())
>>> rule.first_round_.rule_.gross_scores_
{'a': 17, 'b': 16, 'c': 12, 'd': 5, 'e': 0}
>>> rule.second_round_.gross_scores_
{'a': 3, 'b': 2}
```

. . . or changing the elimination algorithm:

```
>>> rule = RuleTwoRound(['a > b > c > d > e', 'b > a > c > d > e', 'c > a > b > d␣
↪> e'],
...                     weights=[2, 2, 1], elimination=EliminationLast(k=-3))
>>> rule.first_round_.rule_.gross_scores_
{'a': 2, 'b': 2, 'c': 1, 'd': 0, 'e': 0}
>>> rule.second_round_.gross_scores_
{'a': 2, 'b': 2, 'c': 1}
```

**cotrailers_**

“Cotrailers” of the election, i.e. the candidates that fare worst in the election. This is the last equivalence class in `order_`. For example, in *RuleScoreNum*, it is the candidates that are tied for the worst score.

**Type** *NiceSet*

**cowinners_**

Cowinners of the election, i.e. the candidates that fare best in the election.. This is the first equivalence class in `order_`. For example, in *RuleScoreNum*, it is the candidates that are tied for the best score.

**Type** *NiceSet*

**elimination_rounds_**

The elimination rounds. A list of *Elimination* objects. All rounds except the last one.

**Type** list

**final_round_**

The final round, which decides the winner of the election.

**Type** *Rule*

**first_round_**

The first round. This is just a shortcut for `self.elimination_rounds_[0]`.

**Type** *Elimination*

**n_candidates_**

Number of candidates.

**Type** int

**rounds_**

The rounds. All rounds but the last one are *Elimination* objects. The last one is a *Rule* object.

### Examples

Note that in some cases, there may be fewer actual rounds than declared in the definition of the rule:

```
>>> rule = RuleSequentialElimination(
...     ['a > b > c > d', 'a > c > d > b', 'a > d > b > c'],
...     rules=[RuleBorda(), RulePlurality(), RulePlurality()],
...     eliminations=[EliminationBelowAverage(), EliminationLast(k=1)])
>>> len(rule.rounds_)
2
>>> rule.elimination_rounds_[0].rule_.gross_scores_
{'a': 9, 'b': 3, 'c': 3, 'd': 3}
>>> rule.final_round_.gross_scores_
{'a': 3}
```

**Type** list

**second_round_**

The second round. This is just an alternative name for `self.final_round_`.

**Type** *Rule*

**strict_order_**

Result of the election as a strict order over the candidates. The first element is the winner, etc. This may use the tie-breaking rule.

**Type** list

**trailer_**

The "trailer" of the election. This is the last candidate in *strict_order_* and also the unfavorable choice of the tie-breaking rule in *cotrailers_*.

**Type** object

**winner_**

The winner of the election. This is the first candidate in *strict_order_* and also the choice of the tie-breaking rule in *cowinners_*.

**Type** object

## 5.8.22 RuleVeto

**class** whalrus.**RuleVeto**(**args, converter: whalrus.converters_ballot.converter_ballot.ConverterBallot = None, scorer: whalrus.scorers.scorer.Scorer = None, **kwargs*)

The veto rule.

**Parameters**

- **args** – Cf. parent class.
- **converter** (ConverterBallot) – Default: *ConverterBallotToVeto*.
- **scorer** (Scorer) – Default: *ScorerVeto*.
- **kwargs** – Cf. parent class.

### Examples

```
>>> RuleVeto(['a', 'b', 'b', 'c', 'c']).winner_
'a'
```

**average_score_**

The average score.

**Type** Number

**average_score_as_float_**

The average score as a float. It is the same as *average_score_*, but converted to a float.

**Type** float

**best_score_as_float_**

The best score as a float. It is the same as *RuleScore.best_score_*, but converted to a float.

**Type** float

**compare_scores**(*one: numbers.Number*, *another: numbers.Number*) → int

Compare two scores.

**Parameters**

- **one** (*object*) – A score.
- **another** (*object*) – A score.

**Returns** 0 if they are equal, a positive number if `one` is greater than `another`, a negative number otherwise.

**Return type** int

**cotrailers_**

“Cotrailers”. The set of candidates with the worst score.

**Type** *NiceSet*

**cowinners_**

Cowinners. The set of candidates with the best score.

**Type** *NiceSet*

**gross_scores_as_floats_**

Gross scores as floats. It is the same as `gross_scores_`, but converted to floats.

**Type** *NiceDict*

**n_candidates_**

Number of candidates.

**Type** int

**scores_as_floats_**

Scores as floats. It is the same as `scores_`, but converted to floats.

**Type** *NiceDict*

**strict_order_**

Result of the election as a strict order over the candidates. The first element is the winner, etc. This may use the tie-breaking rule.

**Type** list

**trailer_**

The "trailer" of the election. This is the last candidate in `strict_order_` and also the unfavorable choice of the tie-breaking rule in `cotrailers_`.

**Type** object

**weights_as_floats_**

Weights as floats. It is the same as `weights_`, but converted to floats.

**Type** *NiceDict*

**winner_**

The winner of the election. This is the first candidate in `strict_order_` and also the choice of the tie-breaking rule in `cowinners_`.

**Type** object

**worst_score_as_float_**

The worst score as a float. It is the same as `RuleScore.worst_score_`, but converted to a float.

**Type** float

## 5.9 Scale

### 5.9.1 Scale

**class** whalrus.**Scale**

A scale used to evaluate the candidates (for `RuleRangeVoting`, `RuleMajorityJudgment`, etc).

This parent class represents a generic scale, where two levels of the scale compare according to their internal methods `__lt__`, `__le__`, etc.

For a subclass, it is sufficient to override the method `lt()` and the other comparison methods will be modified accordingly (assuming it describes a total order).

#### Examples

```
>>> scale = Scale()
>>> scale.lt(1, 7)
True
```

**argsort** (*some_list: list*, *reverse: bool = False*) → list

"Argsort" a list of levels.

**Parameters**

- **some_list** (*list*) – A list of levels.
- **reverse** (*bool*) – If True, then argsort in decreasing order.

**Returns** A list of indexes.

**Return type** list

### Examples

```
>>> Scale().argsort(['a', 'c', 'b'])
[0, 2, 1]
```

**compare**(*one: object*, *another: object*) → int

Compare two levels.

**Parameters**

- **one** (*object*) – A level.
- **another** (*object*) – A level.

**Returns** 0 if they are equal, a positive number if `one` is greater than `another`, a negative number otherwise.

**Return type** int

### Examples

```
>>> Scale().compare('a', 'z')
-1
```

**eq**(*one: object*, *another: object*) → bool

Test "equal". Cf. `lt()`.

**ge**(*one: object*, *another: object*) → bool

Test "greater or equal". Cf. `lt()`.

**gt**(*one: object*, *another: object*) → bool

Test "greater than". Cf. `lt()`.

**high**

The highest element of the scale (or None if the scale is unbounded above).

**Type** object

**le**(*one: object*, *another: object*) → bool

Test "lower or equal". Cf. `lt()`.

**low**

The lowest element of the scale (or None if the scale is unbounded below).

**Type** object

**lt**(*one: object*, *another: object*) → bool

Test "lower than".

Generally, only this method is overridden in the subclasses.

**Parameters**

- **one** (*object*) – A level of the scale.
- **another** (*object*) – A level of the scale.

**Returns** True iff `one` is lower than `another`.

**Return type** bool

### Examples

```
>>> Scale().lt('a', 'z')
True
```

**max** (*iterable: Iterable[T_co]*) → object

Maximum of some levels.

**Parameters** **iterable** (*Iterable*) – An iterable of levels (list, set, etc).

### Examples

```
>>> Scale().max({4, 1, 12})
12
```

**min** (*iterable: Iterable[T_co]*) → object

Minimum of some levels.

**Parameters** **iterable** (*Iterable*) – An iterable of levels (list, set, etc).

### Examples

```
>>> Scale().min({'x', 'a', 'z'})
'a'
```

**ne** (*one: object*, *another: object*) → bool

Test "not equal". Cf. `lt()`.

**sort** (*some_list: list*, *reverse: bool = False*) → None

Sort a list of levels (in place).

**Parameters**

- **some_list** (*list*) – A list of levels.
- **reverse** (*bool*) – If True, then sort in decreasing order.

### Examples

```
>>> some_list = [42, 3, 12]
>>> Scale().sort(some_list)
>>> some_list
[3, 12, 42]
```

## 5.9.2 ScaleFromList

**class** whalrus.**ScaleFromList** (*levels: list*)

Scale derived from a list.

**Parameters** **levels** (*list*) – The list of levels, from the worst to the best.

**argsort** (*some_list: list*, *reverse: bool = False*) → list

### Examples

```
>>> scale = ScaleFromList(['Bad', 'Medium', 'Good', 'Very good', 'Excellent'])
>>> scale.argsort(['Good', 'Bad', 'Excellent'])
[1, 0, 2]
```

**compare**(*one: object*, *another: object*) → int

Compare two levels.

**Parameters**

- **one** (*object*) – A level.
- **another** (*object*) – A level.

**Returns** 0 if they are equal, a positive number if `one` is greater than `another`, a negative number otherwise.

**Return type** int

### Examples

```
>>> Scale().compare('a', 'z')
-1
```

**eq**(*one: object*, *another: object*) → bool

Test "equal". Cf. *lt()*.

**ge**(*one: object*, *another: object*) → bool

Test "greater or equal". Cf. *lt()*.

**gt**(*one: object*, *another: object*) → bool

Test "greater than". Cf. *lt()*.

**high**

```
>>> scale = ScaleFromList(['Bad', 'Medium', 'Good', 'Very good', 'Excellent'])
>>> scale.high
'Excellent'
```

**le**(*one: object*, *another: object*) → bool

Test "lower or equal". Cf. *lt()*.

**low**

```
>>> scale = ScaleFromList(['Bad', 'Medium', 'Good', 'Very good', 'Excellent'])
>>> scale.low
'Bad'
```

**lt**(*one: object*, *another: object*) → bool

```
>>> scale = ScaleFromList(['Bad', 'Medium', 'Good', 'Very good', 'Excellent'])
>>> scale.lt('Medium', 'Excellent')
True
```

**max**(*iterable: Iterable[T_co]*) → object

### Examples

```
>>> scale = ScaleFromList(['Bad', 'Medium', 'Good', 'Very good', 'Excellent'])
>>> scale.max(['Good', 'Bad', 'Excellent'])
'Excellent'
```

**min**(*iterable: Iterable[T_co]*) → object

### Examples

```
>>> scale = ScaleFromList(['Bad', 'Medium', 'Good', 'Very good', 'Excellent'])
>>> scale.min(['Good', 'Bad', 'Excellent'])
'Bad'
```

**ne**(*one: object*, *another: object*) → bool

Test "not equal". Cf. `lt()`.

**sort**(*some_list: list*, *reverse: bool = False*) → None

### Examples

```
>>> scale = ScaleFromList(['Bad', 'Medium', 'Good', 'Very good', 'Excellent'])
>>> some_list = ['Good', 'Bad', 'Excellent']
>>> scale.sort(some_list)
>>> some_list
['Bad', 'Good', 'Excellent']
```

## 5.9.3 ScaleFromSet

**class** whalrus.**ScaleFromSet**(*levels: set*)

Scale derived from a set.

**Parameters** **levels** (*set*) – A set of comparable objects. It is recommended that they are also hashable.

### Examples

Typical usage:

```
>>> scale = ScaleFromSet({-1, 0, 2})
```

A more complex example:

```
>>> class Appreciation:
...     VALUES = {'Excellent': 2, 'Good': 1, 'Medium': 0}
...     def __init__(self, x):
...         self.x = x
...     def __repr__(self):
...         return 'Appreciation(%r)' % self.x
...     def __hash__(self):
...         return hash(self.x)
...     def __lt__(self, other):
```

(continues on next page)

(continued from previous page)

```
...         return Appreciation.VALUES[self.x] < Appreciation.VALUES[other.x]
>>> scale = ScaleFromSet({Appreciation('Excellent'), Appreciation('Good'),
...                      Appreciation('Medium')})
>>> scale.lt(Appreciation('Medium'), Appreciation('Good'))
True
>>> scale.low
Appreciation('Medium')
>>> scale.high
Appreciation('Excellent')
```

**argsort** (*some_list: list*, *reverse: bool = False*) → list

### Examples

```
>>> scale = ScaleFromList(['Bad', 'Medium', 'Good', 'Very good', 'Excellent'])
>>> scale.argsort(['Good', 'Bad', 'Excellent'])
[1, 0, 2]
```

**compare** (*one: object*, *another: object*) → int

Compare two levels.

**Parameters**

- **one** (*object*) – A level.
- **another** (*object*) – A level.

**Returns** 0 if they are equal, a positive number if `one` is greater than `another`, a negative number otherwise.

**Return type** int

### Examples

```
>>> Scale().compare('a', 'z')
-1
```

**eq** (*one: object*, *another: object*) → bool

Test "equal". Cf. *lt()*.

**ge** (*one: object*, *another: object*) → bool

Test "greater or equal". Cf. *lt()*.

**gt** (*one: object*, *another: object*) → bool

Test "greater than". Cf. *lt()*.

**high**

```
>>> scale = ScaleFromList(['Bad', 'Medium', 'Good', 'Very good', 'Excellent'])
>>> scale.high
'Excellent'
```

**le** (*one: object*, *another: object*) → bool

Test "lower or equal". Cf. *lt()*.

**low**

```
>>> scale = ScaleFromList(['Bad', 'Medium', 'Good', 'Very good', 'Excellent'])
>>> scale.low
'Bad'
```

**lt** (*one: object*, *another: object*) → bool

### Examples

```
>>> scale = ScaleFromSet({-1, 0, 2})
>>> scale.lt(0, 2)
True
```

**max** (*iterable: Iterable[T_co]*) → object

### Examples

```
>>> scale = ScaleFromList(['Bad', 'Medium', 'Good', 'Very good', 'Excellent'])
>>> scale.max(['Good', 'Bad', 'Excellent'])
'Excellent'
```

**min** (*iterable: Iterable[T_co]*) → object

### Examples

```
>>> scale = ScaleFromList(['Bad', 'Medium', 'Good', 'Very good', 'Excellent'])
>>> scale.min(['Good', 'Bad', 'Excellent'])
'Bad'
```

**ne** (*one: object*, *another: object*) → bool

Test "not equal". Cf. *lt()*.

**sort** (*some_list: list*, *reverse: bool = False*) → None

### Examples

```
>>> scale = ScaleFromList(['Bad', 'Medium', 'Good', 'Very good', 'Excellent'])
>>> some_list = ['Good', 'Bad', 'Excellent']
>>> scale.sort(some_list)
>>> some_list
['Bad', 'Good', 'Excellent']
```

## 5.9.4 ScaleInterval

**class** whalrus.**ScaleInterval** (*low: numbers.Number = 0*, *high: numbers.Number = 1*)

A scale given by a continuous interval of numbers.

**Parameters**

- **low** (*Number*) – Lowest grade.

- **high** (*Number*) – Highest grade.

### Examples

```
>>> ScaleInterval(low=0, high=2.5)
ScaleInterval(low=0, high=Fraction(5, 2))
```

**argsort** (*some_list: list*, *reverse: bool = False*) → list

#### Examples

```
>>> ScaleInterval(low=0, high=1).argsort([.3, .1, .7])
[1, 0, 2]
```

**compare** (*one: object*, *another: object*) → int

Compare two levels.

**Parameters**

- **one** (*object*) – A level.
- **another** (*object*) – A level.

**Returns** 0 if they are equal, a positive number if `one` is greater than `another`, a negative number otherwise.

**Return type** int

#### Examples

```
>>> Scale().compare('a', 'z')
-1
```

**eq** (*one: object*, *another: object*) → bool

Test "equal". Cf. `lt()`.

**ge** (*one: object*, *another: object*) → bool

Test "greater or equal". Cf. `lt()`.

**gt** (*one: object*, *another: object*) → bool

Test "greater than". Cf. `lt()`.

**high**

#### Examples

```
>>> ScaleInterval(low=0, high=1).high
1
```

**le** (*one: object*, *another: object*) → bool

Test "lower or equal". Cf. `lt()`.

**low**

### Examples

```
>>> ScaleInterval(low=0, high=1).low
0
```

**lt**(*one: object*, *another: object*) → bool
Test "lower than".

Generally, only this method is overridden in the subclasses.

**Parameters**

- **one** (*object*) – A level of the scale.
- **another** (*object*) – A level of the scale.

**Returns** True iff `one` is lower than `another`.

**Return type** bool

### Examples

```
>>> Scale().lt('a', 'z')
True
```

**max**(*iterable: Iterable[T_co]*) → object

### Examples

```
>>> ScaleInterval(low=0, high=1).max([.3, .1, .7])
0.7
```

**min**(*iterable: Iterable[T_co]*) → object

### Examples

```
>>> ScaleInterval(low=0, high=1).min([.3, .1, .7])
0.1
```

**ne**(*one: object*, *another: object*) → bool
Test "not equal". Cf. `lt()`.

**sort**(*some_list: list*, *reverse: bool = False*) → None

### Examples

```
>>> some_list = [.3, .1, .7]
>>> ScaleInterval(low=0, high=1).sort(some_list)
>>> some_list
[0.1, 0.3, 0.7]
```

### 5.9.5 ScaleRange

**class** whalrus.**ScaleRange**(*low: int*, *high: int*)

A scale of consecutive integers.

Remark: for a scale of non-consecutive integers, such as {-1, 0, 2}, use the *ScaleFromSet*.

**Parameters**

- **low** (*int*) – Lowest integer.
- **high** (*int*) – Highest integer.

#### Examples

```
>>> scale = ScaleRange(low=0, high=5)
```

**argsort**(*some_list: list*, *reverse: bool = False*) → list

#### Examples

```
>>> ScaleRange(low=0, high=5).argsort([3, 1, 4])
[1, 0, 2]
```

**compare**(*one: object*, *another: object*) → int

Compare two levels.

**Parameters**

- **one** (*object*) – A level.
- **another** (*object*) – A level.

**Returns** 0 if they are equal, a positive number if one is greater than another, a negative number otherwise.

**Return type** int

#### Examples

```
>>> Scale().compare('a', 'z')
-1
```

**eq**(*one: object*, *another: object*) → bool

Test "equal". Cf. *lt()*.

**ge**(*one: object*, *another: object*) → bool

Test "greater or equal". Cf. *lt()*.

**gt**(*one: object*, *another: object*) → bool

Test "greater than". Cf. *lt()*.

**high**

### Examples

```
>>> ScaleRange(low=0, high=5).high
5
```

**le** (*one: object*, *another: object*) → bool

Test "lower or equal". Cf. `lt()`.

**low**

### Examples

```
>>> ScaleRange(low=0, high=5).low
0
```

**lt** (*one: object*, *another: object*) → bool

Test "lower than".

Generally, only this method is overridden in the subclasses.

**Parameters**

- **one** (`object`) – A level of the scale.
- **another** (`object`) – A level of the scale.

**Returns** True iff `one` is lower than `another`.

**Return type** bool

### Examples

```
>>> Scale().lt('a', 'z')
True
```

**max** (*iterable: Iterable[T_co]*) → object

### Examples

```
>>> ScaleRange(low=0, high=5).max([3, 1, 4])
4
```

**min** (*iterable: Iterable[T_co]*) → object

### Examples

```
>>> ScaleRange(low=0, high=5).min([3, 1, 4])
1
```

**ne** (*one: object*, *another: object*) → bool

Test "not equal". Cf. `lt()`.

**sort** (*some_list: list*, *reverse: bool = False*) → None

#### Examples

```
>>> some_list = [3, 1, 4]
>>> ScaleRange(low=0, high=5).sort(some_list)
>>> some_list
[1, 3, 4]
```

## 5.10 Scorer

### 5.10.1 Scorer

**class** whalrus.**Scorer**(**args, scale: whalrus.scales.scale.Scale = None, **kwargs*)

A "scorer".

A `Scorer` is a callable whose inputs are a ballot, a voter and a set of candidates (the set of candidates of the election). When the scorer is called, it loads its arguments. The output of the call is the scorer itself. But after the call, you can access to the computed variables (ending with an underscore), such as `scores_`.

At the initialization of a `Scorer` object, some options can be given, such as a scale. In some subclasses, there can be some additional options.

**Parameters**

- **args** – If present, these parameters will be passed to `__call__` immediately after initialization.
- **scale** (`Scale`) – The scale in which scores are computed.
- **kwargs** – If present, these parameters will be passed to `__call__` immediately after initialization.

**ballot_**

This attribute stores the ballot given in argument of the `__call__`.

**Type** *Ballot*

**voter_**

This attribute stores the voter given in argument of the `__call__`.

**Type** object

**candidates_**

This attribute stores the candidates given in argument of the `__call__`.

**Type** *NiceSet*

#### Examples

Cf. `ScorerLevels` for some examples.

**scores_**

The scores. To each candidate, this dictionary associates either a level in the scale or None. For the meaning of None, cf. `RuleRangeVoting` for example. Intuitively: a score of 0 means that the value 0 is counted in the average, whereas None is not counted at all (i.e. the weight of the voter is not even counted in the denominator when computing the average).

**Type** *NiceDict*

**scores_as_floats_**

The scores, given as floats. It is the same as *scores_*, but converted to floats.

Like all conversions to floats, it is advised to use this attribute for display purposes only. For computation, you should always use *scores_*, which usually manipulates fractions and therefore allows for exact computation.

**Raises** `ValueError` – If the scores cannot be converted to floats.

**Type** *NiceDict*

### 5.10.2 ScorerBorda

**class** whalrus.**ScorerBorda**(**args*, *absent_give_points: bool = True*, *absent_receive_points: Optional[bool] = True*, *unordered_give_points: bool = True*, *unordered_receive_points: Optional[bool] = True*, ***kwargs*)

A Borda scorer for `BallotOrder`.

**Parameters**

- **args** – Cf. parent class.
- **absent_give_points** (*bool*) – Whether absent candidates give points to the other candidates.
- **absent_receive_points** (*bool or None*) – Whether absent candidates receives points. Remark: 0 means that any absent candidate receives the score 0 (which will be counted in its average Borda score, median Borda score, etc); in contrast, None means that the absent candidate receives no score (hence this voter will be excluded from the computation of its average Borda score, median Borda score, etc).
- **unordered_give_points** (*bool*) – Whether unordered candidates give points to the ordered candidates, i.e. they are considered as being in a lower position in the ranking.
- **unordered_receive_points** (*bool or None*) – Whether unordered candidates receive points. Like for `absent_receive_points`, None means that an unordered candidate receives no score at all.
- **kwargs** – Cf. parent class.

#### Examples

Typical usage:

```
>>> ScorerBorda(ballot=BallotOrder('a > b > c'), voter='Alice',
...             candidates={'a', 'b', 'c'}).scores_
{'a': 2, 'b': 1, 'c': 0}
```

In the example below, candidates *a*, *b* and *c* are “ordered”, *d* and *e* are “unordered”, and *f* and *g* are “absent” in the ballot, meaning that these candidates were not even available when the voter cast her ballot. The options allows for different ways to take these special cases into account:

```
>>> ballot = BallotOrder('a > b ~ c', candidates={'a', 'b', 'c', 'd', 'e'})
>>> candidates_election = {'a', 'b', 'c', 'd', 'e', 'f', 'g'}
>>> ScorerBorda(ballot, candidates=candidates_election).scores_as_floats_
{'a': 6.0, 'b': 4.5, 'c': 4.5, 'd': 2.5, 'e': 2.5, 'f': 0.5, 'g': 0.5}
>>> ScorerBorda(ballot, candidates=candidates_election,
...             absent_receive_points=False).scores_as_floats_
```

(continues on next page)

(continued from previous page)

```
{'a': 6.0, 'b': 4.5, 'c': 4.5, 'd': 2.5, 'e': 2.5, 'f': 0.0, 'g': 0.0}
>>> ScorerBorda(ballot, candidates=candidates_election,
...             absent_receive_points=False, absent_give_points=False).scores_as_
↪floats_
{'a': 4.0, 'b': 2.5, 'c': 2.5, 'd': 0.5, 'e': 0.5, 'f': 0.0, 'g': 0.0}
>>> ScorerBorda(ballot, candidates=candidates_election,
...             absent_receive_points=False, absent_give_points=False,
...             unordered_receive_points=False).scores_as_floats_
{'a': 4.0, 'b': 2.5, 'c': 2.5, 'd': 0.0, 'e': 0.0, 'f': 0.0, 'g': 0.0}
>>> ScorerBorda(ballot, candidates=candidates_election,
...             absent_receive_points=False, absent_give_points=False,
...             unordered_receive_points=False, unordered_give_points=False).
↪scores_as_floats_
{'a': 2.0, 'b': 0.5, 'c': 0.5, 'd': 0.0, 'e': 0.0, 'f': 0.0, 'g': 0.0}
```

Usage of None in the options:

```
>>> ScorerBorda(ballot, candidates=candidates_election,
...             absent_receive_points=None, unordered_receive_points=None).scores_
↪as_floats_
{'a': 6.0, 'b': 4.5, 'c': 4.5}
```

**scores_as_floats_**

The scores, given as floats. It is the same as `scores_`, but converted to floats.

Like all conversions to floats, it is advised to use this attribute for display purposes only. For computation, you should always use `scores_`, which usually manipulates fractions and therefore allows for exact computation.

**Raises** `ValueError` – If the scores cannot be converted to floats.

**Type** *NiceDict*

### 5.10.3 ScorerBucklin

**class** whalrus.**ScorerBucklin**(**args, k: int = 1, unordered_receive_points: Optional[bool] = True, absent_receive_points: Optional[bool] = True, **kwargs*)

Scorer for Bucklin's rule.

**Parameters**

- **args** – Cf. parent class.
- **k** (*int*) – The number of points to distribute. Intuitively: the `k` candidates at the highest ranks will receive 1 point each. In case of tie, some points may be divided between the tied candidates (see below).
- **unordered_receive_points** (*bool or None.*) – Whether unordered candidates should receive points (see below).
- **absent_receive_points** (*bool or None.*) – Whether absent candidates should receive points (see below).
- **kwargs** – Cf. parent class.

#### Examples

Typical usage:

```
>>> ScorerBucklin(BallotOrder('a > b > c > d > e'),
...               candidates={'a', 'b', 'c', 'd', 'e'}, k=2).scores_
{'a': 1, 'b': 1, 'c': 0, 'd': 0, 'e': 0}
```

In the example below, candidates *a*, *b* and *c* are "ordered", *d* and *e* are "unordered", and *f* and *g* are "absent" in the ballot, meaning that they were not even available when the voter cast her ballot. By default, we count as if the unordered candidates were below the ordered candidates, and the absent candidates even lower:

```
>>> ballot = BallotOrder('a > b ~ c', candidates={'a', 'b', 'c', 'd', 'e'})
>>> candidates_election = {'a', 'b', 'c', 'd', 'e', 'f', 'g'}
>>> ScorerBucklin(ballot, candidates=candidates_election, k=2).scores_as_floats_
{'a': 1.0, 'b': 0.5, 'c': 0.5, 'd': 0.0, 'e': 0.0, 'f': 0.0, 'g': 0.0}
>>> ScorerBucklin(ballot, candidates=candidates_election, k=4).scores_as_floats_
{'a': 1.0, 'b': 1.0, 'c': 1.0, 'd': 0.5, 'e': 0.5, 'f': 0.0, 'g': 0.0}
>>> ScorerBucklin(ballot, candidates=candidates_election, k=6).scores_as_floats_
{'a': 1.0, 'b': 1.0, 'c': 1.0, 'd': 1.0, 'e': 1.0, 'f': 0.5, 'g': 0.5}
```

Using the options, unordered and/or absent candidates can always receive 0 point, or even not be mentioned in the score dictionary at all:

```
>>> ScorerBucklin(ballot, candidates=candidates_election, k=6,
...     unordered_receive_points=False, absent_receive_points=None).scores_
{'a': 1, 'b': 1, 'c': 1, 'd': 0, 'e': 0}
```

**scores_as_floats_**

The scores, given as floats. It is the same as `scores_`, but converted to floats.

Like all conversions to floats, it is advised to use this attribute for display purposes only. For computation, you should always use `scores_`, which usually manipulates fractions and therefore allows for exact computation.

**Raises** `ValueError` – If the scores cannot be converted to floats.

**Type** *NiceDict*

### 5.10.4 ScorerLevels

**class** whalrus.**ScorerLevels**(**args*, *level_ungraded: object = None*, *level_absent: object = None*, ***kwargs*)

A standard scorer for :class:`BallotLevel`.

**Parameters**

- **args** – Cf. parent class.
- **level_ungraded** (*object*) – The level of the scale used for ungraded candidates, or None.
- **level_absent** (*object*) – The level of the scale used for absent candidates, or None.
- **kwargs** – Cf. parent class.

#### Examples

In the most general syntax, firstly, you define the scorer:

```
>>> scorer = ScorerLevels(level_absent=0)
```

Secondly, you use it as a callable to load some particular arguments:

```
>>> scorer(ballot=BallotLevels({'a': 10, 'b': 7, 'c': 3}), voter='Alice',
...        candidates={'a', 'b', 'c', 'd'})  # doctest:+ELLIPSIS
<... object at ...>
```

Finally, you can access the computed variables:

```
>>> scorer.scores_
{'a': 10, 'b': 7, 'c': 3, 'd': 0}
```

Later, if you wish, you can load other arguments (ballot, etc) with the same scorer, and so on.

Optionally, you can specify arguments as soon as the *Scorer* object is initialized. This allows for "one-liners" such as:

```
>>> ScorerLevels(ballot=BallotLevels({'a': 10, 'b': 7, 'c': 3}), voter='Alice',
...              candidates={'a', 'b', 'c', 'd'}, level_absent=0).scores_
{'a': 10, 'b': 7, 'c': 3, 'd': 0}
```

In the example below, candidates *a*, *b* and *c* are "ordered", *d* is "unordered", and *e* is "absent" in the ballot, meaning that *e* were not even available when the voter cast her ballot. The options of the scorer provide different ways to take these special cases into account:

```
>>> ballot=BallotLevels({'a': 10, 'b': 7, 'c': 3}, candidates={'a', 'b', 'c', 'd'}
↪)
>>> candidates_election = {'a', 'b', 'c', 'd', 'e'}
>>> ScorerLevels(ballot, candidates=candidates_election).scores_
{'a': 10, 'b': 7, 'c': 3}
>>> ScorerLevels(ballot, candidates=candidates_election,
...              level_ungraded=-5).scores_
{'a': 10, 'b': 7, 'c': 3, 'd': -5}
>>> ScorerLevels(ballot, candidates=candidates_election,
...              level_ungraded=-5, level_absent=-10).scores_
{'a': 10, 'b': 7, 'c': 3, 'd': -5, 'e': -10}
```

**scores_as_floats_**

The scores, given as floats. It is the same as `scores_`, but converted to floats.

Like all conversions to floats, it is advised to use this attribute for display purposes only. For computation, you should always use `scores_`, which usually manipulates fractions and therefore allows for exact computation.

**Raises** `ValueError` – If the scores cannot be converted to floats.

**Type** *NiceDict*

### 5.10.5 ScorerPlurality

**class** whalrus.**ScorerPlurality**(**args*, *count_abstention: bool = False*, ***kwargs*)

A Plurality scorer for *BallotPlurality*.

**Parameters**

- **args** – Cf. parent class.
- **count_abstention** (*bool*) – If False (default), then an abstention grants no score at all. If True, then an abstention gives 0 point to each candidate (cf. below).
- **kwargs** – Cf. parent class.

### Examples

Typical usage:

```
>>> ScorerPlurality(BallotPlurality('a'), candidates={'a', 'b', 'c'}).scores_
{'a': 1, 'b': 0, 'c': 0}
```

Using the option `count_abstention`:

```
>>> ScorerPlurality(BallotPlurality(None), candidates={'a', 'b', 'c'}).scores_
{}
>>> ScorerPlurality(BallotPlurality(None), candidates={'a', 'b', 'c'},
...                 count_abstention=True).scores_
{'a': 0, 'b': 0, 'c': 0}
```

**`scores_as_floats_`**

The scores, given as floats. It is the same as `scores_`, but converted to floats.

Like all conversions to floats, it is advised to use this attribute for display purposes only. For computation, you should always use `scores_`, which usually manipulates fractions and therefore allows for exact computation.

**Raises** `ValueError` – If the scores cannot be converted to floats.

**Type** *NiceDict*

## 5.10.6 ScorerPositional

**class** `whalrus.`**`ScorerPositional`**(**args, points_scheme: list = None, points_fill: Optional[numbers.Number] = 0, points_unordered: Optional[numbers.Number] = 0, points_absent: Optional[numbers.Number] = None, **kwargs*)

A positional scorer for strict order ballots.

**Parameters**

- **`args`** – Cf. parent class.
- **`points_scheme`** (*`list`*) – The list of points to be attributed to the (first) candidates of a ballot.
- **`points_fill`** (*`Number or None`*) – Points for ordered candidates that have a rank beyond the `points_scheme`.
- **`points_unordered`** (*`Number or None`*) – Points for the unordered candidates.
- **`points_absent`** (*`Number or None`*) – Points for the absent candidates.
- **`kwargs`** – Cf. parent class.

### Examples

The top candidate in the ballot receives `points_scheme[0]` points, the second one receives `points_scheme[1]` points, etc:

```
>>> ScorerPositional(ballot=BallotOrder('a > b > c'), points_scheme=[10, 5, 3]).
↪scores_
{'a': 10, 'b': 5, 'c': 3}
```

The points scheme does not need to have the same length as the ballot:

```
>>> ScorerPositional(ballot=BallotOrder('a > b > c'), points_scheme=[3, 2, 1, .
↪5]).scores_
{'a': 3, 'b': 2, 'c': 1}
>>> ScorerPositional(ballot=BallotOrder('a > b > c'), points_scheme=[3, 2]).
↪scores_
{'a': 3, 'b': 2, 'c': 0}
```

A typical usage of this is k-Approval voting:

```
>>> ScorerPositional(ballot=BallotOrder('a > b > c > d > e'), points_scheme=[1,␣
↪1]).scores_
{'a': 1, 'b': 1, 'c': 0, 'd': 0, 'e': 0}
```

In the example below, candidates *a*, *b* and *c* are "ordered", *d* is "unordered", and *e* is "absent" in the ballot, meaning that *e* was not even available when the voter cast her ballot. The options of the scorer provide different ways to take these special cases into account:

```
>>> ballot=BallotOrder('a > b > c', candidates={'a', 'b', 'c', 'd'})
>>> candidates_election = {'a', 'b', 'c', 'd', 'e'}
>>> ScorerPositional(ballot, candidates=candidates_election, points_scheme=[3,␣
↪2]).scores_
{'a': 3, 'b': 2, 'c': 0, 'd': 0}
>>> ScorerPositional(ballot, candidates=candidates_election, points_scheme=[3, 2],
...     points_fill=-1, points_unordered=-2, points_absent=-3).scores_
{'a': 3, 'b': 2, 'c': -1, 'd': -2, 'e': -3}
>>> ScorerPositional(ballot, candidates=candidates_election, points_scheme=[3, 2],
...     points_fill=None, points_unordered=None, points_absent=None).scores_
{'a': 3, 'b': 2}
```

**`scores_as_floats_`**

The scores, given as floats. It is the same as `scores_`, but converted to floats.

Like all conversions to floats, it is advised to use this attribute for display purposes only. For computation, you should always use `scores_`, which usually manipulates fractions and therefore allows for exact computation.

**Raises** `ValueError` – If the scores cannot be converted to floats.

**Type** *NiceDict*

### 5.10.7 ScorerVeto

**class** `whalrus.`**`ScorerVeto`**(**args*, *count_abstention: bool = False*, ***kwargs*)

A Veto scorer for `BallotVeto`.

**Parameters**

- **`args`** – Cf. parent class.
- **`count_abstention`** (`bool`) – If False (default), then an abstention grants no score at all. If True, then an abstention gives 0 point to each candidate (cf. below).
- **`kwargs`** – Cf. parent class.

### Examples

Typical usage:

```
>>> ScorerVeto(BallotVeto('a'), candidates={'a', 'b', 'c'}).scores_
{'a': -1, 'b': 0, 'c': 0}
```

Using the option `count_abstention`:

```
>>> ScorerVeto(BallotVeto(None), candidates={'a', 'b', 'c'}).scores_
{}
>>> ScorerVeto(BallotVeto(None), candidates={'a', 'b', 'c'},
...                   count_abstention=True).scores_
{'a': 0, 'b': 0, 'c': 0}
```

**`scores_as_floats_`**

The scores, given as floats. It is the same as `scores_`, but converted to floats.

Like all conversions to floats, it is advised to use this attribute for display purposes only. For computation, you should always use `scores_`, which usually manipulates fractions and therefore allows for exact computation.

**Raises** `ValueError` – If the scores cannot be converted to floats.

**Type** *NiceDict*

## 5.11 Util Module

**class** `whalrus.utils.utils.`**`DeleteCacheMixin`**

Mixin used to delete cached properties.

Cf. decorator *`cached_property()`*.

### Examples

```
>>> class Example(DeleteCacheMixin):
...     @cached_property
...     def x(self):
...         print('Big computation...')
...         return 6 * 7
>>> a = Example()
>>> a.x
Big computation...
42
>>> a.x
42
>>> a.delete_cache()
>>> a.x
Big computation...
42
```

**class** `whalrus.utils.utils.`**`NiceDict`**

A dict that prints in the order of the keys (when they are comparable).

### Examples

```
>>> my_dict = NiceDict({'b': 51, 'a': 42, 'c': 12})
>>> my_dict
{'a': 42, 'b': 51, 'c': 12}
```

**class** whalrus.utils.utils.**NiceSet**

A set that prints in order (when the elements are comparable).

### Examples

```
>>> my_set = NiceSet({'b', 'a', 'c'})
>>> my_set
{'a', 'b', 'c'}
```

whalrus.utils.utils.**cached_property**(*f*)

Decorator used in replacement of @property to put the value in cache automatically.

The first time the attribute is used, it is computed on-demand and put in cache. Later accesses to the attributes will use the cached value.

Cf. *DeleteCacheMixin* for an example.

whalrus.utils.utils.**convert_number**(*x: numbers.Number*)

Try to convert a number to a fraction (or an integer).

**Parameters** **x** (*Number*) –

**Returns** x, trying to convert it into a fraction (or an integer).

**Return type** Number

### Examples

```
>>> convert_number(2.5)
Fraction(5, 2)
>>> convert_number(2.0)
2
```

whalrus.utils.utils.**dict_to_items**(*d: dict*) → list

Convert a dict to a list of pairs (key, value).

**Parameters** **d** (*dict*) –

**Returns** The result is similar to d.items(), but if the keys are comparable, they appear in ascending order.

**Return type** list of pairs

### Examples

```
>>> dict_to_items({'b': 2, 'c': 0, 'a': 1})
[('a', 1), ('b', 2), ('c', 0)]
```

whalrus.utils.utils.**dict_to_str**(*d: dict*) → str

Convert dict to string.

**Parameters** **d** (*dict*) –

**Returns** The result is similar to str(d), but if the keys are comparable, they appear in ascending order.

**Return type** str

### Examples

```
>>> dict_to_str({'b': 2, 'c': 0, 'a': 1})
"{'a': 1, 'b': 2, 'c': 0}"
```

whalrus.utils.utils.**my_division**(*x: numbers.Number*, *y: numbers.Number*, *divide_by_zero: numbers.Number = None*)

Division of two numbers, trying to be exact if it is reasonable.

**Parameters**

- **x** (*Number*) –
- **y** (*Number*) –
- **divide_by_zero** (*Number*) – The value to be returned in case of division by zero. If None (default), then it raises a ZeroDivisionError.

**Returns** The division of *x* by *y*.

**Return type** Number

### Examples

```
>>> my_division(5, 2)
Fraction(5, 2)
```

If *x* or *y* is a float, then the result is a float:

```
>>> my_division(Fraction(5, 2), 0.1)
25.0
>>> my_division(0.1, Fraction(5, 2))
0.04
```

If *x* and *y* are integers, decimals or fractions, then the result is a fraction:

```
>>> my_division(2, Fraction(5, 2))
Fraction(4, 5)
>>> my_division(Decimal('0.1'), Fraction(5, 2))
Fraction(1, 25)
```

You can specify a particular return value in case of division by zero:

```
>>> my_division(1, 0, divide_by_zero=42)
42
```

whalrus.utils.utils.**parse_weak_order**(*s: str*) → list

Convert a string representing a weak order to a list of sets.

**Parameters** **s** (*str*) –

**Returns** A list of sets, where each set is an indifference class. The first set of the list contains the top (= most liked) candidates, while the last set of the list contains the bottom (= most disliked) candidates.

**Return type** list

### Examples

```
>>> s = 'Alice ~ Bob ~ Catherine32 > me > you ~ us > them'
>>> parse_weak_order(s) == [{'Alice', 'Bob', 'Catherine32'}, {'me'}, {'you', 'us'}
↪, {'them'}]
True
```

`whalrus.utils.utils.`**`set_to_list`**(*s: set*) → list

Convert a set to a list.

**Parameters** **`s`** (*`set`*) –

**Returns** The result is similar to list(s), but if the elements of the set are comparable, they appear in ascending order.

**Return type** list

### Examples

```
>>> set_to_list({2, 42, 12})
[2, 12, 42]
```

`whalrus.utils.utils.`**`set_to_str`**(*s: set*) → str

Convert a set to a string.

**Parameters** **`s`** (*`set`*) –

**Returns** The result is similar to str(s), but if the elements of the set are comparable, they appear in ascending order.

**Return type** str

### Examples

```
>>> set_to_str({2, 42, 12})
'{2, 12, 42}'
```

`whalrus.utils.utils.`**`take_closest`**(*my_list*, *my_number*)

In a list, take the closest element to a given number.

From https://stackoverflow.com/questions/12141150/from-list-of-integers-get-number-closest-to-a-given-value .

**Parameters**

- **`my_list`** (*`list`*) – A list sorted in ascending order.
- **`my_number`** (*`Number`*) –

**Returns** The element of `my_list` that is closest to `my_number`. If two numbers are equally close, return the smallest number.

**Return type** Number

### Examples

```
>>> take_closest([0, 5, 10], 3)
5
```

# CHAPTER 6

# Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

## 6.1 Types of Contributions

### 6.1.1 Report Bugs

Report bugs at https://github.com/francois-durand/whalrus/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

### 6.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with “bug” and “help wanted” is open to whoever wants to implement it.

### 6.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with “enhancement” and “help wanted” is open to whoever wants to implement it.

### 6.1.4 Write Documentation

Whalrus could always use more documentation, whether as part of the official Whalrus docs, in docstrings, or even on the web in blog posts, articles, and such.

### 6.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/francois-durand/whalrus/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

## 6.2 Get Started!

Ready to contribute? Here's how to set up *whalrus* for local development.

1. Fork the *whalrus* repo on GitHub.
2. Clone your fork locally:

   ```
   $ git clone git@github.com:your_name_here/whalrus.git
   ```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

   ```
   $ mkvirtualenv whalrus
   $ cd whalrus/
   $ python setup.py develop
   ```

4. Create a branch for local development:

   ```
   $ git checkout -b name-of-your-bugfix-or-feature
   ```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

   ```
   $ flake8 whalrus tests
   $ python setup.py test or py.test
   $ tox
   ```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

   ```
   $ git add .
   $ git commit -m "Your detailed description of your changes."
   $ git push origin name-of-your-bugfix-or-feature
   ```

7. Submit a pull request through the GitHub website.

## 6.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.7, 3.4, 3.5 and 3.6, and for PyPy. Check https://travis-ci.org/francois-durand/whalrus/pull_requests and make sure that the tests pass for all supported Python versions.

## 6.4 Tips

To run a subset of tests:

```
$ py.test tests.test_whalrus
```

## 6.5 Deploying

A reminder for the maintainers on how to deploy. Make sure all your changes are committed (including an entry in HISTORY.rst). Then run:

```
$ bumpversion patch # possible: major / minor / patch
$ git push
$ git push --tags
```

Travis will then deploy to PyPI if tests pass.

## 6.6 Useful links

https://github.com/francois-durand/whalrus https://readthedocs.org/projects/whalrus/builds/

# CHAPTER 7

# Credits

## 7.1 Development Lead

- François Durand <fradurand@gmail.com>

## 7.2 Contributors

None yet. Why not be the first?

CHAPTER 8

# History

## 8.1 0.4.6 (2020-12-01): Improve test coverage

- Reach 100% of test coverage. Cf. https://codecov.io/gh/francois-durand/whalrus.
- Convert documentation to Numpy style. The documentation is not changed much in html format, but is more readable in plain text.
- Remove hash function for `BallotOneName` and `BallotOrder`. It had a bug, and fixing it would have implied to change all sets of candidates to frozen sets. Since this function is non-essential, we decided to remove it instead.
- Fix bug in `MatrixWeightedMajority` when using the option `ordered_vs_absent` or `absent_vs_ordered`.
- Fix bug in `Rule.trailer_` when there is only one candidate in the election.

## 8.2 0.4.5 (2020-11-26): Fix Missing Files in Deployment

- Files from some sub-packages, such as `scale`, were missing. This release fixes that bug.

## 8.3 0.4.4 (2020-11-26): Fix PyPI deployment

- Fix PyPI deployment.

## 8.4 0.4.3 (2020-11-26): GitHub Actions

- This patch concerns Whalrus' developpers only. To develop and maintain the package, it uses GitHub actions instead of additional services such as Travis-CI and ReadTheDocs.

- Use Codecov.
- Prepare support for Numpy documentation style (not used yet).
- Prepare support for notebooks in documentation (not used yet).

## 8.5 0.4.2 (2019-08-22): Speeding Up

- Minor patch to speed up the computation of the winner in some cases.

## 8.6 0.4.1 (2019-04-01): Tie-breaking

- Fix a bug related to random tie-break.
- In the arguments of class `RuleRankedPairs`, the tie-break can be given directly, instead of having to go through the argument `matrix`.

## 8.7 0.4.0 (2019-03-29): Schulze

- Implement Schulze rule.

## 8.8 0.3.0 (2019-03-29): Ranked Pairs

- Implement Ranked Pairs rule.

## 8.9 0.2.1 (2019-03-28): Optimize argument passing

- Optimize argument passing between child classes, their parent classes and their `__call__` function.

## 8.10 0.2.0 (2019-03-21): Classic voting systems

- First “real” release, where most classic voting systems are implemented.

## 8.11 0.1.0 (2018-03-13): First release

- First release on PyPI.

CHAPTER 9

# Indices and tables

- genindex
- modindex
- search

# Python Module Index

# Index