Wiggelen Documentation

Release 0.4.1

Leiden University Medical Center, Martijn Vermaat, Jeroen Laros

December 22, 2015

Contents

Wiggelen is a Python library for working with wiggle tracks (WIG files). It also provides a command line interface to some of its functionality.

The main goal of Wiggelen is to provide light-weight and unified access to wiggle tracks.

```
>>> import wiggelen
>>> for x in wiggelen.walk(open('test.wig')):
...     print 'chr%s:%d\t%s' % x
...
chr18:34344  629.0
chr18:34345  649.0
chr18:34446  657.0
chrM:308     520.0
chrM:309     519.0
```

CHAPTER 1

User documentation

New users should probably start here.

1.1 Installation

The Wiggelen source code is hosted on GitHub. Supported Python versions for running Wiggelen are 2.6, 2.7, 3.2, 3.3, and PyPy (unit tests are run automatically on these platforms using the Travis CI service). Wiggelen can be installed either via the Python Package Index (PyPI) or from the source code.

1.1.1 Latest release via PyPI

To install the latest release via PyPI using pip:

```
pip install wiggelen
```

1.1.2 Development version

You can also clone and use the latest development version directly from the GitHub repository:

```
git clone https://github.com/martijnvermaat/wiggelen.git
cd wiggelen
python setup.py install
```

1.2 User guide

Wiggelen is a light-weigh library and tries not to over-engineer. For example, builtin datatypes such as tuples are used instead of custom objects. Sane defaults are used throughout and things like indices are handled transparently to the user.

The central operation in Wiggelen is walking a track. Be it in `fixedSteps` or `variableSteps` format, using any window size and step interval, walking a track yields values one position at a time. Many operations accept walkers as input and/or return walkers as output.

This guide uses `a.wig` and `b.wig` as example wiggle tracks, with the following contents, respectively:

```
track type=wiggle_0 name=a visibility=full
variableStep chrom=MT
1 520.0
2 536.0
3 553.0
4 568.0
```

```
track type=wiggle_0 name=b visibility=full
variableStep chrom=MT
1 510.0
2 512.0
5 508.0
8 492.0
```

1.2.1 Walking over a track

Walking a track is done with the *wiggelen.walk()* function, which yields tuples of *region*, *position*, *value*:

```
>>> for region, position, value in walk(open('a.wig')):
...     print region, position, value
...
MT 1 520.0
MT 2 536.0
MT 3 553.0
MT 4 568.0
```

Note: Walkers are implemented as generators, therefore walking (i.e. iterating) over them means consuming them. In other words, you can only iterate over a walker once.

Multiple tracks can be walked simultaneously with the *wiggelen.zip_()* function, yielding a walker with lists of values for each track:

```
>>> a = walk(open('a.wig'))
>>> b = walk(open('b.wig'))
>>> for region, position, value in zip_(a, b):
...     print region, position, value
...
1 1 [520.0, 510.0]
1 2 [536.0, 512.0]
1 3 [553.0, None]
1 4 [568.0, None]
1 5 [None, 508.0]
1 8 [None, 492.0]
```

Sometimes it is useful to force a walk over every subsequent position, even when some positions are skipped in the original track file. This can be done with the *wiggelen.fill()* function:

```
>>> for region, position, value in fill(walk(open('b.wig'))):
...     print region, position, value
...
1 1 510.0
1 2 512.0
1 3 None
1 4 None
1 5 508.0
1 6 None
```

```
1 7 None
1 8 492.0
```

1.2.2 Writing a walker to a track

Any walker can be written to a track file using the `wiggelen.write()` function, which by default writes to standard output:

```
>>> write(walk(open('a.wig')), name='My example')
track type=wiggle_0 name="My example"
variableStep chrom=MT
1 520.0
2 536.0
3 553.0
4 568.0
```

1.2.3 Value transformations

For doing simple transformations on values from a walker, the `itertools.imap()` function is often useful:

```
>>> from itertools import imap
>>> transform = lambda (r, p, v): (r, p, v * 2)
>>> for region, position, value in imap(transform,
...                                     walk(open('a.wig'))):
...     print region, position, value
...
MT 1 1040.0
MT 2 1072.0
MT 3 1106.0
MT 4 1136.0
```

Similarly, the `itertools.ifilter()` function can be used to quickly filter some values from a walker.

The `wiggelen.transform` module contains several predefined transformations for calculating the derivative of a walker:

```
>>> for region, position, value in transform.forward_divided_difference(
...   walk(open('a.wig'))):
...     print region, position, value
...
MT 1 16.0
MT 2 17.0
MT 3 15.0
```

Note: Walker values can be of any type, but valid wiggle tracks according to the specification can only contain *int* or *float* values.

1.2.4 Coverage intervals

Genomic intervals of consecutively defined positions can be extracted from a walker using the `wiggelen.intervals.coverage()` function:

```
>>> for region, begin, end in intervals.coverage(walk(open('b.wig'))):
...     print region, begin, end
...
MT 1 2
MT 5 5
MT 8 8
```

1.2.5 Merging walkers

The *wiggelen.merge* module provides a way to merge any number of wiggle tracks with a given merge operation. Some standard merge operations are pre-defined in *wiggelen.merge.mergers*.

```
>>> for region, position, value in merge.merge(
...   walk(open('a.wig')), walk(open('b.wig')),
...   merger=merge.mergers['sum']):
...     print region, position, value
...
1 1 1030.0
1 2 1048.0
1 3 553.0
1 4 568.0
1 5 508.0
1 8 492.0
```

1.2.6 Distance matrices

Wiggelen can calculate the distance between two or more wiggle tracks according to a pairwise multiset distance metric. This is implemented in the *wiggelen.distance* module and can be used to assess similarity of next generation datasets.

```
>>> distance.distance(open('a.wig'), open('b.wig'))
{(1, 0): 0.5704115928792818}
```

Four pairwise multiset distance metrics are pre-defined in *wiggelen.distance.metrics*.

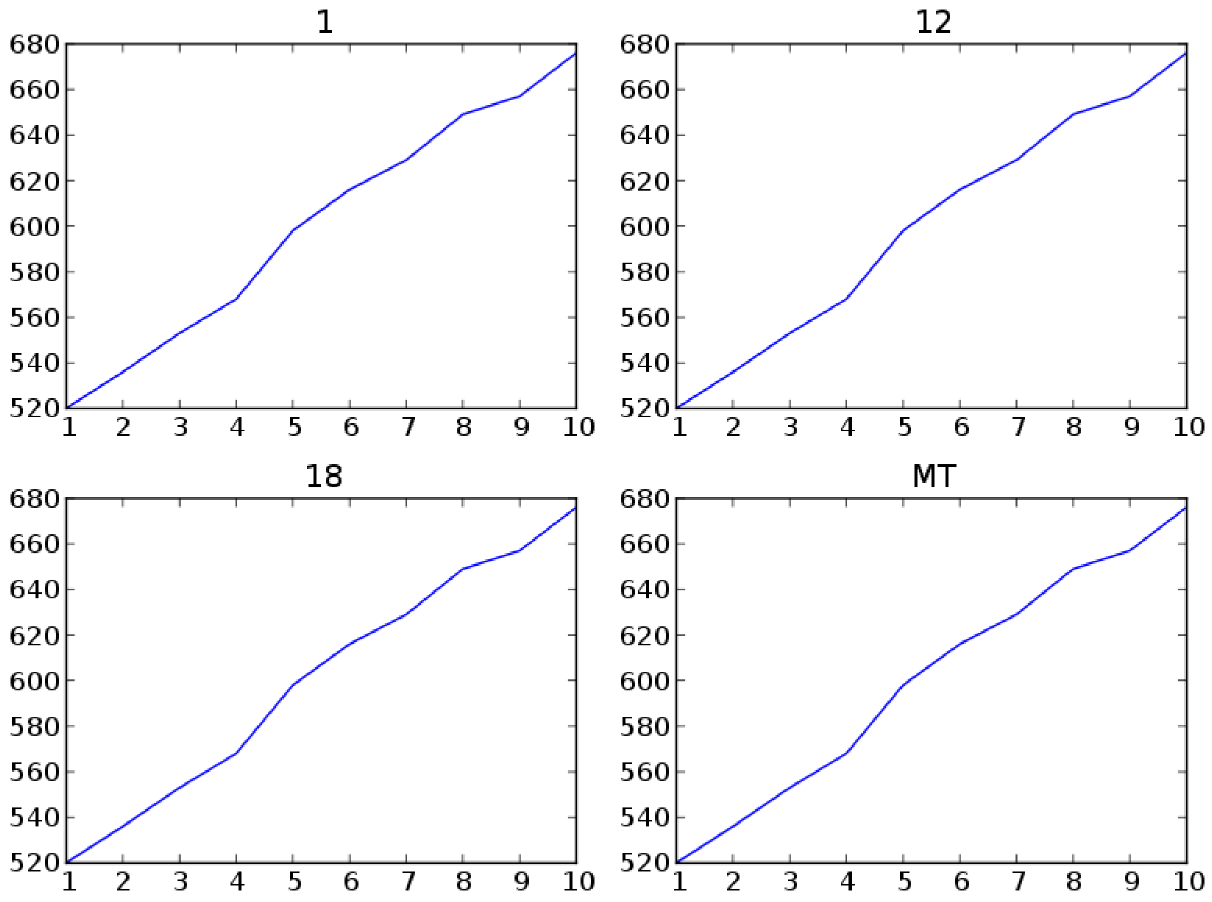
1.2.7 Plotting tracks

Some rudimentary functionality for plotting a wiggle track is provided by the `wiggelen.plot` module. It requires the matplotlib package to be installed.

Note: The `wiggelen.plot.plot()` function should not be used on very large tracks.

For example, to quickly visualize the `tests/data/complex.wig` file in the Wiggelen source repository:

```
>>> fig, _, _, _ = plot.plot(walk(open('tests/data/complex.wig')))
>>> fig.show()
```



1.3 Command line interface

Some of the functionality in Wiggelen is provided through a simple command line interface.

Since the average scientist is too lazy to write complete documentation, you'll just find a quick dump of the command line help output below.

```
martijn@hue:~$ wiggelen -h
usage: wiggelen [-h]
                {index,sort,scale,fill,derivative,plot,coverage,merge,distance} ...

Wiggelen command line interface.

optional arguments:
  -h, --help            show this help message and exit

subcommands:
  {index,sort,scale,derivative,plot,merge,distance}
                        subcommand help
    index               build index for wiggle track
    sort                sort wiggle track regions alphabetically
    scale               scale values in a wiggle track
    fill                fill undefined positions in a wiggle track
    derivative          create derivative of a wiggle track
    plot                visualize wiggle tracks in a plot (requires
                        matplotlib)
    coverage            create coverage BED track of a wiggle track
    merge               merge any number of wiggle tracks in various ways
    distance            calculate the distance between wiggle tracks
```

Well, I guess nobody ever got fired for showing a quick example, so here you go:

```
martijn@hue:~$ wiggelen distance tests/data/*.wig
A: tests/data/a.wig
B: tests/data/b.wig
C: tests/data/complex.wig
D: tests/data/c.wig
E: tests/data/empty.wig

      A     B     C     D     E
A     x
B   0.687   x
C   0.000 0.687   x
D   0.901 0.958 0.901   x
E   0.974 0.952 0.974 0.748   x
```

CHAPTER 2

API reference

Documentation on a specific function, class or method can be found in the API reference.

2.1 API reference

2.1.1 wiggelen

Wiggelen, working with wiggle tracks in Python.

The wiggle (WIG) format is for storing dense, continuous genomic data such as GC percent, probability scores, read depth, and transcriptome data.

exception `wiggelen.`**`ParseError`**

Raised if a wiggle track cannot be parsed.

exception `wiggelen.`**`ReadError`**

Raised if a wiggle track does not provide random access. Reading with random access is needed for using and creating an index.

`wiggelen.`**`walk`**`(`*track=<open file '<stdin>', mode 'r'>*, *force_index=False*`)`

Walk over the track and yield (region, position, value) tuples.

The values are always of type *int* or *float*.

Parameters

- **`track`** (*file*) – Wiggle track.
- **`force_index`** (*bool*) – Force creating an index if it does not yet exist.

Returns Tuples of (region, position, value) per defined position.

Return type generator(str, int, _)

Example:

```
>>> for x in walk():
...     x
...
('chr18', 34344, 629.0)
('chr18', 34345, 649.0)
('chr18', 34446, 657.0)
('chrM',  308,   520.0)
('chrM',  309,   519.0)
```

`wiggelen.`**`zip_`**`(`**walkers*`)`

Walk over all tracks simultaneously and for each position yield the region, position and a list of values for each track, or *None* in case the track has no value on the position.

Note: This assumes the order of regions is compatible over all walkers. If you are unsure if this is the case for your input wiggle tracks, use the `walk()` function with the *force_index* keyword argument.

Parameters **`walkers`** (*list(generator(str, int, _))*) – List of generators yielding tuples of (region, position, value) per defined position.

Returns Tuples of (region, position, values) per defined position.

Return type generator(str, int, list(_))

Example:

```
>>> for x in zip_(walk(open('a.wig')), walk(open('b.wig'))):
...     x
...
('18', 7, [29.0, None])
('18', 8, [49.0, None])
('18', 9, [None, 87.0])
('MT', 1, [20.0, None])
('MT', 2, [36.0, 92.0])
```

`wiggelen.`**`fill`**`(`*walker*, *regions=None*, *filler=None*, *only_edges=False*`)`

Fill in undefined positions with *filler* (or *None*).

Parameters

- **`walker`** (*generator(str, int, _)*) – Tuple of (region, position, value) per defined position.
- **`regions`** (*dict(str, (int, int))*) – Dictionary with regions as keys and (start, stop) tuples as values. If not *None*, fill positions from start to stop (both including) in these regions. If *None*, fill positions in all regions between their first and last defined positions.
- **`filler`** – Value to use for filling undefined positions.
- **`only_edges`** (*bool*) – Only fill the first and last of continuously undefined positions.

Returns Tuples of (region, position, value) per position where value is *filler* if it was not defined in the original walker.

Return type generator(str, int, _)

Example:

```
>>> for x in walk(open('a.wig')):
...     x
...
('MT', 3, 29.0)
('MT', 5, 49.0)
('MT', 8, 87.0)
('MT', 9, 20.0)
>>> for x in fill(walk(open('a.wig')):
...     x
...
('MT', 3, 29.0)
('MT', 4, None)
('MT', 5, 49.0)
('MT', 6, None)
```

```
    ('MT', 7, None)
    ('MT', 8, 87.0)
    ('MT', 9, 20.0)
```

The *only_edges* argument might seem a bit out of place here, but can be useful in combination with *filler=0* when creating a line plot. Without any filling, non-zero lines may be plotted where there is actually no data.

Note: This might be a tiny bit memory-hungry on Python 2.x if there are *very* large gaps to fill since we use the range function to generate the positions. I don't think it's worth it to add version specific code paths for this.

wiggelen.**write**(*walker*, *track=<open file '<stdout>', mode 'w'>*, *serializer=<type 'str'>*, *name=None*, *description=None*)

Write items from a walker to a wiggle track.

Parameters

- **walker** (*generator(str, int, _)*) – Tuples of (region, position, value) per defined position.
- **track** (*file*) – Writable file handle.
- **serializer** (*function(_ -> str)*) – Function making strings from values.
- **name** (*str*) – Optional track name (displayed to the left of the track in the UCSC Genome Browser).
- **description** (*str*) – Optional track description (displayed as center label in the UCSC Genome Browser).

Note: Values of *None* are discarded.

Example:

```
>>> write(walk(open('a.wig')), name='My example')
track type=wiggle_0 name="My example"
variableStep chrom=1
1 520.0
4 536.0
8 553.0
variableStep chrom=MT
1 568.0
2 598.0
6 616.0
```

2.1.2 wiggelen.merge

Merge any number of wiggle tracks in various ways.

The algorithm can be parameterized by a merge operation. Four of these operations are predefined in *mergers*:

Merger `sum`: Compute the sum of all values.

Merger `mean`: Compute the mean of all values (and use 0 for undefined values).

Merger `count`: Compute the number of defined values.

Merger `minus`: Subtract the second value from the first (and use 0 for undefined values). Only defined on exactly two values.

Merger `min`: Compute the minimum of all values (and use 0 for undefined values).

Merger `max`: Compute the maximum of all values (and use 0 for undefined values).

Merger `div`: Divide the second value by the first (and use 0 for undefined values). Only defined on exactly two values.

Merger `intersect`: Return the first value is the second value is defined and non-zero (and use 0 for undefined values). Only defined on exactly two values.

Merger `ctz`: Select the value closest to 0. (and use 0 if there is a mix of positive and negative values).

`wiggelen.merge.`**`merge`**`(`**walkers*, ***options*`)`

Merge wiggle tracks.

This assumes the walkers have their regions in the same order. You can force this by using indices. Example:

```
>>> from wiggelen import walk
>>> walkers = [walk(open(track), force_index=True)
...            for track in ('a.wig', 'b.wig', 'c.wig')]
>>> for x in merge(*walkers):
...     x
...
('18', 8, 849.0)
('18', 9, 987.0)
('MT', 1, 820.0)
```

Parameters

- **`walkers`** (*list(generator(str, int, _))*) – List of generators yielding tuples of (region, position, value) per defined position.
- **`merger`** (*function(list(_) -> _)*) – Merge operation (default: sum).

Returns Tuples of (region, position, merged value) per defined position in *walkers*.

Return type generator(str, int, _)

`wiggelen.merge.`**`mergers`** **= {'count': <function <lambda> at 0x7fa9f037fb18>, 'div': <function <lambda> at 0x7fa9f037fc**

Predefined mergers. See *`wiggelen.merge`* for their definition.

2.1.3 wiggelen.distance

Calculate the distance between two wiggle tracks using a metric designed for multisets.

This module can be used to assess similarity of next generation sequencing datasets. A multiset distance measure is used for pairwise comparison of genomic information as provided by wiggle tracks.

The algorithm can be parameterized by a pairwise distance metric. Four of these metrics are predefined in *`metrics`*:

Metric `a`: $\frac{|x-y|}{(x+1)(y+1)}$

Metric `b`: $\frac{|x-y|}{x+y+1}$

Metric `c`: $\frac{\max(x,y)\,|x-y|}{(x^2+1)(y^2+1)}$

Metric `d`: $\frac{|x-y|}{\max(x,y)+1}$

Note: These metrics are ill-defined on the interval (0, 1) so we scale all values if necessary.

`wiggelen.distance.`**`distance`**`(`**tracks*, ***options*`)`

Calculate the pairwise distances between wiggle tracks.

Parameters

- **tracks** – List of wiggle tracks.
- **metric** – Pairwise distance metric (default: a).
- **threshold** (*float*) – Threshold for noise filter (default: no noise filter)

Returns Pairwise distances between *tracks* as a mapping from coordinates in the distance matrix to their values.

Return type dict((int, int), float)

`wiggelen.distance.`**`matrix`**(*size*, *reflexive=False*, *symmetric=False*)

Create all coordinates in a square matrix.

With the default *False* value for *reflexive* and *symmetric*, include only the coordinates below the diagonal.

Parameters

- **size** (*int*) – Width and height of the matrix.
- **reflexive** (*bool*) – Include coordinates on (*x*, *x*) diagonal.
- **symmetric** (*bool*) – Include coordinates (*x*, *y*) above the diagonal (where $x < y$).

Returns All coordinates in the matrix as tuples.

Return type list(int, int)

Examples:

```
>>> matrix(5)
[(1, 0),
 (2, 0), (2, 1),
 (3, 0), (3, 1), (3, 2),
 (4, 0), (4, 1), (4, 2), (4, 3)]
>>> matrix(5, reflexive=True)
[(0, 0),
 (1, 0), (1, 1),
 (2, 0), (2, 1), (2, 2),
 (3, 0), (3, 1), (3, 2), (3, 3),
 (4, 0), (4, 1), (4, 2), (4, 3), (4, 4)]
>>> matrix(5, symmetric=True)
[        (0, 1), (0, 2), (0, 3), (0, 4),
 (1, 0),         (1, 2), (1, 3), (1, 4),
 (2, 0), (2, 1),         (2, 3), (2, 4),
 (3, 0), (3, 1), (3, 2),         (3, 4),
 (4, 0), (4, 1), (4, 2), (4, 3)        ]
```

`wiggelen.distance.`**`metrics`** **= {'a': <function <lambda> at 0x7fa9f037f668>, 'c': <function <lambda> at 0x7fa9f037f320**

Predefined pairwise distance metrics. See `wiggelen.distance` for their definition.

`wiggelen.distance.`**`normalize`**(**values*)

Normalize values relative to the smallest value.

Parameters **values** (*list(float)*) – List of values.

Returns Scale the values such that the minimum is 1.

Return type list(float)

2.1.4 wiggelen.transform

Various transformations on wiggle tracks.

`wiggelen.transform.`**`backward_divided_difference`**`(`*walker*, *step=None*, *auto_step=False*`)`

Derivative calculated by the backward divided difference method.

Note: This transformation only works on walkers with numerical values.

Parameters

- **`walker`** (*generator(str, int, float)*) – Generator yielding tuples of (region, position, value) per defined position.
- **`step`** (*int*) – Restrict calculation to positions that are this far apart (no restriction if *None*).
- **`auto_step`** (*bool*) – If *True* and *step=None*, automatically set *step* to a value based on the first two positions in *walker*.

Returns Tuple of (region, position, derivative value) per defined position in *walker* for which the derivative value is defined.

Return type generator(str, int, float)

`wiggelen.transform.`**`central_divided_difference`**`(`*walker*, *step=None*`)`

Derivative calculated by the central divided difference method.

Note: This transformation only works on walkers with numerical values.

Parameters

- **`walker`** (*generator(str, int, float)*) – Generator yielding tuples of (region, position, value) per defined position.
- **`step`** (*int*) – Restrict calculation to positions that are this far apart. If *None*, automatically set *step* to a value based on the first two positions in *walker*.

Returns Tuple of (region, position, derivative value) per defined position in *walker* for which the derivative value is defined.

Return type generator(str, int, float)

`wiggelen.transform.`**`forward_divided_difference`**`(`*walker*, *step=None*, *auto_step=False*`)`

Derivative calculated by the forward divided difference method.

Note: This transformation only works on walkers with numerical values.

Parameters

- **`walker`** (*generator(str, int, float)*) – Generator yielding tuples of (region, position, value) per defined position.
- **`step`** (*int*) – Restrict calculation to positions that are this far apart (no restriction if *None*).
- **`auto_step`** (*bool*) – If *True* and *step=None*, automatically set *step* to a value based on the first two positions in *walker*.

Returns Tuple of (region, position, derivative value) per defined position in *walker* for which the derivative value is defined.

Return type generator(str, int, float)

2.1.5 wiggelen.intervals

Get covered intervals from wiggle tracks and write to BED format.

`wiggelen.intervals.`**`coverage`**`(`*walker*`)`

Get intervals of consecutively defined positions from a walker.

Parameters **`walker`** (*generator(str, int, _)*) – Tuple of *(region, position, value)* per defined position.

Returns Tuples of *(region, begin, end)* per position where *begin* and *end* are one-based and inclusive.

Return type generator(str, int, int)

Example:

```
>>> for x in coverage(walk(open('a.wig'))):
...     x
...
('MT', 3, 3)
('MT', 5, 20)
('MT', 400, 420)
```

`wiggelen.intervals.`**`write`**`(`*intervals*, *track=<open file '<stdout>', mode 'w'>*, *name=None*, *description=None*`)`

Write intervals to a bed track.

Parameters

- **`intervals`** (*generator(str, int, int)*) – Tuples of (region, begin, end) per interval.
- **`track`** (*file*) – Writable file handle.
- **`name`** (*str*) – Optional track name (displayed to the left of the track in the UCSC Genome Browser).
- **`description`** (*str*) – Optional track description (displayed as center label in the UCSC Genome Browser).

Example:

```
>>> write(coverage(walk(open('a.wig'))), name='My example')
track name="My example"
MT 2 3
MT 4 20
MT 399 420
```

2.1.6 wiggelen.plot

2.1.7 wiggelen.index

Index regions/chromosomes in wiggle tracks for random access.

Indexing a wiggle track results in a mapping of regions to summaries. The summaries are dictionaries including start and stop positions and some statistical metrics. A summary for the entire wiggle track is included as the special region `_all`.

This data can be written to a file next to the wiggle track file (in case this is a regular file). Example of the serialization we use:

```
region=_all,start=0,stop=12453,sum=4544353,count=63343
region=1,start=47,stop=3433,sum=4353,count=643
region=X,start=3433,stop=8743,sum=454,count=343
region=Y,start=8743,stop=10362,sum=7353,count=343
region=MT,start=10362,stop=12453,sum=353,count=143
```

Note that we do not impose a certain order on the lines in the index nor on the fields on a line.

Additional custom fields can be added to the index by providing custom field definitions. Such a definition is created with the *Field* constructor and the following arguments:

- The name of the field.
- A function casting a field value from *string*.
- Initial value.
- Aggregate function used as the function argument in a reduce- or fold-like operation to construct the field value. This function takes as inputs the accumulated field value, the current value and the current span, and returns a new accumulated field value.

As an example, the standard *sum* field could be defined as the following tuple:

```
Field('sum', float, 0, lambda acc, value, span: acc + value * span)
```

In practice, choose unique names for custom fields, not clashing with the standard fields such as *sum*.

`wiggelen.index.`**`CACHE_INDEX`** **= True**
: Whether or not indices are cached in memory during execution.

class `wiggelen.index.`**`Field`**(*name*, *caster*, *init*, *func*)
: Type for custom index field definitions.

  `caster`
  : Alias for field number 1

  `func`
  : Alias for field number 3

  `init`
  : Alias for field number 2

  `name`
  : Alias for field number 0

`wiggelen.index.`**`INDEX_SUFFIX`** **= '.idx'**
: Suffix used for index files.

`wiggelen.index.`**`WRITE_INDEX`** **= True**
: Whether or not indices are written to a file.

`wiggelen.index.`**`clear_cache`**()
: Clear the in-memory cache of index objects.

`wiggelen.index.`**`index`**(*track=<open file '<stdin>', mode 'r'>*, *force=False*, *fields=None*)
: Return index of region positions in track.

  Parameters

  - **`track`** (*file*) – Wiggle track.
  - **`force`** (*bool*) – Force creating an index if it does not yet exist.
  - **`fields`** (*list*) – List of custom index field definitions.

Returns Wiggle track index and index filename.

Return type dict(str, dict(str, _)), str

`wiggelen.index.`**`read_index`**(*track=<open file '<stdin>', mode 'r'>*, *fields=None*)

Try to read the index from a file.

Parameters

- **`track`** (*file*) – Wiggle track the index belongs to.
- **`fields`** (*list*) – List of custom index field definitions.

Returns Wiggle track index, or *None* if the index could not be read.

Return type dict(str, dict(str, _))

`wiggelen.index.`**`write_index`**(*idx*, *track=<open file '<stdout>', mode 'w'>*)

Try to write the index to a file and return its filename.

Parameters

- **`idx`** (*dict(str, dict(str, _))*) – Wiggle track index.
- **`track`** (*file*) – Wiggle track the index belongs to.

Returns Filename for the written index, or *None* if the index could not be written.

Return type str

CHAPTER 3

Additional notes

3.1 Development

Development of Wiggelen happens on GitHub: https://github.com/martijnvermaat/wiggelen

3.1.1 Contributing

Contributions to Wiggelen are very welcome! They can be feature requests, bug reports, bug fixes, unit tests, documentation updates, or anything els you may come up with.

3.1.2 Coding style

In general, try to follow the PEP 8 guidelines for Python code and PEP 257 for docstrings.

3.1.3 Unit tests

To run the unit tests with nose, just run `nosetests -v`.

3.1.4 Versioning

A normal version number takes the form X.Y.Z where X is the major version, Y is the minor version, and Z is the patch version. Development versions take the form X.Y.Z.dev where X.Y.Z is the closest future release version.

Note that this scheme is not 100% compatible with SemVer which would require X.Y.Z-dev instead of X.Y.Z.dev but compatibility with setuptools is more important for us. Other than that, version semantics are as described by SemVer.

Releases are published at PyPI and available from the GitHub git repository as tags.

Release procedure

Releasing a new version is done as follows:

1. Make sure the section in the `CHANGES` file for this release is complete and there are no uncommitted changes.

   Note: Commits since release X.Y.Z can be listed with `git log vX.Y.Z..` for quick inspection.

2. Update the `CHANGES` file to state the current date for this release and edit `wiggelen/__init__.py` by updating *__date__* and removing the `dev` value from *__version_info__*.

   Commit and tag the version update:

```
git commit -am 'Bump version to X.Y.Z'
git tag -a 'vX.Y.Z'
git push --tags
```

3. Upload the package to PyPI:

```
python setup.py sdist upload
```

4. Add a new entry at the top of the `CHANGES` file like this:

```
Version X.Y.Z+1
---------------

Release date to be decided.
```

   Increment the patch version and add a `dev` value to *__version_info__* in `wiggelen/__init__.py` and commit these changes:

```
git commit -am 'Open development for X.Y.Z+1'
```

3.1.5 Todo

These are some general todo notes. More specific notes can be found by grepping the source code for `Todo`.

- Option to specify region(s) to use from a track, in that order.
- Beter unit tests coverage.
- Profile code to identify what's keeping us from doing stuff fast.
- Fill optionally takes a BED file of regions to fill, but it will only consider one entry per chromosome (and this is not clearly documented). There may also be other cases where a dictionary of *chromosome->(start, stop)* is used where we perhaps want to generalize to *chromosome->list(start, stop)* (or *list(chromosome, start, stop)*, or an OrderedMultiDict).

3.2 Changelog

Here you can see the full list of changes between each Wiggelen release.

3.2.1 Version 0.4.1

Released on May 3rd 2014.

- New `--share-y` and `--y-lim` options for `wiggelen plot` command.
- Command line or importable definition of custom merge function for `wiggelen merge` command.
- Optionally only report posititions in regions defined by the `--genome` argument for `wiggelen fill` command.

3.2.2 Version 0.4.0

Released on February 17th 2014.

- Command line interface to *wiggelen.fill*.
- Fix off-by-one error in reading genome file argument in `wiggelen plot` command.
- New merger functions *ctz* (*c*losest *t*o *z*ero), and *max* (thanks Jeroen F.J. Laros).

3.2.3 Version 0.3.0

Released on November 29th 2013.

- New merger functions *min*, *div*, and *intersect* (thanks Jeroen F.J. Laros).
- Discard *None* values when writing.

3.2.4 Version 0.2.0

Released on September 2nd 2013.

- Auto scale tracks in distance calculation if needed (the metrics in *wiggelen.distance* are not defined on the (0, 1) interval).
- Add min,posmin,max fields to track index.
- Optionally consider edges in *wiggelen.fill* (*only_edges* argument).
- Introduce the *plot* module for visualizing tracks.
- Rename command line `visualize` function to `plot`.

3.2.5 Version 0.1.6

Released on August 8th 2013.

- Support *fixedStep* definitions without *step* argument. This is not valid by the spec, so a practical consideration. Fixes issue #1.

3.2.6 Version 0.1.5

Released on July 21st 2013.

- Optional track name (*-n* or *–name*) and description (*-d* or *–description*) for command line functions.
- Command line argument *-n* or *–no-indices* for *merge* has been renamed to *-x* or *–no-indices*.
- Slight performance improvement in parsing.

3.2.7 Version 0.1.4

Released on July 15th 2013.

- Fix parsing tracks in fixedStep mode.
- Intervals BED track is now correctly written tab-delimited.

- Fix distance module on Python 3.2+.

3.2.8 Version 0.1.3

Released on June 11th 2013.

- Fix distance calculation on values below threshold. Positions where both values are below the given threshold are ignored.

3.2.9 Version 0.1.2

Released on May 6th 2013.

- Coverage intervals. Module *intervals* implements extraction of coverage intervals from walkers and writing them to BED files. Coverage intervals with an optional value threshold can be written for any wiggle track to a BED file using the command line *coverage* command.
- Add name and description arguments to *write* (Jeroen Laros).
- Fix dependencies declaration in setup.py (only affects Python 2.6).

3.2.10 Version 0.1.1

Released on April 27th 2013.

- Fix installation from PyPi (`README.rst` was missing in sdist).

3.2.11 Version 0.1.0

Released on April 26th 2013.

First public release.

3.3 Copyright

Wiggelen is licensed under the MIT License, meaning you can do whatever you want with it as long as all copies include these license terms. The full license text can be found below.

3.3.1 Authors

Wiggelen is written and maintained by Martijn Vermaat at Leiden University Medical Center and includes contributions by Jeroen Laros.

- Leiden University Medical Center <humgen@lumc.nl>
- Martijn Vermaat <martijn@vermaat.name>
- Jeroen Laros <j.f.j.laros@vermaat.name>

3.3.2 License

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index