VIS Framework Documentation

Release 2.0.0

Christopher Antila

September 08, 2015

VIS

Music Analysis Framework

This is the documentation for the VIS music analysis framework, made available as free software according to the terms of the Affero General Public Licence, version 3 or any later. VIS is a Python package that uses the music21 and pandas libraries to build a ridiculously flexible and preposterously easy system for writing computer music analysis programs. The developers hope to lower the barrier to empirical music analysis to the point that North American music theorists feel compelled to use techniques common among scientists since the 1980s. The VIS framework was produced by the McGill University team of the ELVIS Project. Learn more about the project at our website, http://elvisproject.ca.

CHAPTER 1

Who Should Read This Documentation

If you are a programmer who wishes to use the `vis` Python objects directly, to use the framework as part of another project, or to add functionality to your own instance of the `vis` Web app, you should read this document.

If you are not a programmer, this document is probably not for you, since we assume readers are fluent with Python. You may still wish to read the *Design Principles* section to get an idea of how to use our framework, but it is still quite technical. If you wish to learn about computer-driven music analysis, if you are looking for help using the ELVIS counterpoint Web app, or if you are looking for research findings produced by the ELVIS Project, you will find these resources on the ELVIS website when they become available.

CHAPTER 2

What This Documentation Covers

The VIS API documentation explains the framework's architecture, how to install and use the framework, and how to use the built-in analyzer modules.

How to Use This Documentation

If you are new to VIS, we recommend you read *Design Principles* and continue through both tutorials.

If you forgot the basics, we recommend you read this paragraph. The VIS framework uses two models (`IndexedPiece` and `AggregatedPieces`) to fetch results for one or multiple pieces, respectively. Use `metadata()` to fetch metadata and `get_data()` to run analyses and fetch results. Call `get_data()` with a list of analyzers to run, and a dictionary with the settings they use. Learn what an analyzer does, what it requires for input, and what settings it uses by reading the class documentation. Learn the output format of an analyzer by reading the `run()` documentation.

If you have a question about an analyzer, learn what it does, what it requires for input, and what settings it uses by reading the class documentation. Learn the output format by reading the `run()` documentation.

Important: Remember to have a lot of fun.

Use the `vis` Framework

4.1 Design Principles

The vis framework has a simple design: write an `analyzer` to make an analytic judgment about a piece, then use the built-in `models` to ensure analyzers are run in the right order, with the right inputs and settings. Since music analysis is a complex task (really, a complicated complex of tasks), and the vis framework is highly abstracted, our simple design requires much explanation.

4.1.1 Write an Analyzer

There are two types of analyzers: indexers and experimenters. Indexers take a `music21.stream.Score` or the result of another indexer, perform a calculation, then produce output that can sensibly be attached to a specific moment of a piece. Indexers may be relatively simple, like the *`IntervalIndexer`*, which accepts an index of the notes and rests in a piece and outputs an index of the intervals between all possible part pairs. Indexers may be complicated, like the *`NGramIndexer`*, which accepts at least one index of anything, and outputs an index of successions found therein. An indexer might tell you the scale degrees in a part, or the harmonic function of chords in a score.

Experimenters always accept the result of an indexer or another experimenter, perform a calculation, then produce results that cannot be sensibly attached to a specific moment of a piece. Experimenters may be relatively simple, like the *`FrequencyExperimenter`*, which accepts any index and counts the number of occurrences of unique objects found within. Experimenters may be complicated, like one that accepts the result of the `FrequencyExperimenter`, then outputs a Markov transition model.

The vis framework ships analyzers for various tasks, but we think most users will extend the framework with their own analyzers.

4.1.2 Use a Model

The vis framework has two data models. Use *`IndexedPiece`* for a single piece and *`AggregatedPieces`* for pieces as a group. In a typical application, you will write analyzers but never use them directly, and never modify but always use the models. The models know how to run analyzers on the piece or pieces they hold, how to import pieces safely and efficiently, and how to find and access metadata. In the future, the models may support storing results from analyzers in a database so they need not be recalculated for future analyses, use multiprocessing to speed up analyses on multi-core computers, or facilitate transit to and from analyzers in other languages like Haskell. We recommend you use the models to benefit from new features without modifying your programs, since they (should not) change the API.

4.1.3 How to Start

After you install the framework, we recommend you begin with the two tutorials below (refer to *Tutorial: Make a new Workflow* and *Tutorial: Use the WorkflowManager*). When you wish to write a new analyzer, refer to the documentation and source code for the `TemplateIndexer` and `TemplateExperimenter`.

4.2 Install and Test the Framework

4.2.1 Install for Deployment

You must install the VIS Framework before you use it. If you will not write extensions for the Framework, you may use `pip` to install the package from the Python Package Index (PyPI—https://pypi.python.org/pypi/vis-framework/). Run this command:

```
$ pip install vis-framework
```

You may also wish to install some or all of the optional dependencies:

- `numexpr` and `bottleneck`, which speed up `pandas`.
- `openpyxl`, which allows `pandas` to export Excel-format spreadsheets.
- `cython` and `tables`, which allow `pandas` to export HDF5-format binary files.

You may install optional dependencies in the same ways as VIS itself. For example:

```
$ pip install numexpr bottleneck
```

4.2.2 Install for Development

If you wish to install the VIS Framework for development work, we recommend you clone our Git repository from https://github.com/ELVIS-Project/vis/, or even make your own fork on GitHub. You may also wish to checkout a particular version for development with the "checkout" command, as `git checkout tags/vis-framework-1.2.3` or `git checkout master`.

If you installed git, but you need help to clone a repository, you may find useful information in the git documentation.

After you clone the vis repository, you should install its dependencies (currently music21, pandas, and mock), for which we recommend you use `pip`. From the main VIS directory, run `pip install -r requirements.txt` to automatically download and install the library dependencies as specified in our `requirements.txt` file. We also recommend you run `pip install -r optional_requirements.txt` to install several additional packages that improve the speed of pandas and allow additional output formats (Excel, HDF5). You may need to use `sudo` or `su` to run pip with the proper privileges. If you do not have pip installed, use your package manager (the package is probably called `python-pip`—at least for users of Fedora, Ubuntu, and openSUSE). If you are one of those unfortunate souls who uses Windows, or worse, Mac OS X, then clearly we come from different planets. The pip documentation may help you.

During development, you should usually start `python` (or `ipython`, etc.) from within the main "vis" directory to ensure proper importing.

After you install the VIS Framework, we recommend you run our automated tests. From the main vis directory, run `python run_tests.py`. Python prints `.` for every test that passes, and a large error or warning for every test that fails. Certain versions of music21 may cause tests to fail; refer to *Known Issues and Limitations* for more information.

The `WorkflowManager` is not required for the framework's operation. We recommend you use the `WorkflowManager` directly or as an example to write new applications. The vis framework gives you tools to answer a wide variety of musical questions. The `WorkflowManager` uses the framework to answer specific questions. Please refer to *Tutorial: Use the WorkflowManager* for more information. If you will not use the `WorkflowManager`, we recommend you delete it: remove the `workflow.py` and `other_tests/test_workflow.py` files.

4.2.3 Install R and ggplot2 for Graphs

If you wish to produce graphs with the VIS Framework, you must install an R interpreter and the "ggplot2" library. We use the version 3.0.x series of R.

If you use a "Windows" or "OS X" computer, download a pre-compiled binary from http://cran.r-project.org. If you use a "Linux" computer (or "BSD," etc.), check your package manager for R 3.0.x. You may have a problem if you search for "R," since it is a common letter, so we recommend you assume the package is called "R" and try to search only if that does not work. If your distribution does not provide an R binary, or provides an older version than 3.0.0, install R from source code: http://cran.r-project.org/doc/manuals/r-release/R-admin.html.

In all cases, if you encounter a problem, the R manuals are extensive, but require careful attention.

Your distribution may provide a package for "ggplot" or "ggplot2." The following instructions work for all operating systems:

1. Start R (with superuser privileges, if not on Windows).
2. Run the following command to install ggplot:

```
install.packages("ggplot2")
```

3. Run the following commands to test R and ggplot:

```
huron <- data.frame(year=1875:1972, level=as.vector(LakeHuron))
library(plyr)
huron$decade <- round_any(huron$year, 10, floor)
library(ggplot)
h <- ggplot(huron, aes(x=year))
h + geom_ribbon(aes(ymin=level-1, ymax=level+1))
```

Expect to see a chart like this:

Quit R. You do not need to save your workspace:

```
q()
```

4.3 Known Issues and Limitations

- Issue: music21 releases earlier than specified in our `requirements.txt` file may contain bugs that cause some of our automated tests to fail. We recommend you use a music21 release from the 1.7.x series. In particular, we do not recommend versions 1.5.0 or 1.6.0.
- Issue: Additionally, music21 releases later than 1.7.1 have not been tested. We do not recommend using music21 1.8.0 or later until after the VIS Framework 2.0 release.
- Limitation: By default, the vis framework does not use multiprocessing at all. If you install the optional packages for pandas, many of the pandas-based indexers and experimenters will use multi-threading in C. However, there are many opportunities to use multiprocessing where we have yet to do so. While we initially planned for the indexers and experimenters to use multiprocessing, we later decided that the high overhead of multiprocessing in Python means we should leave the multiprocessing implementation up to application developers—the realm of the *`WorkflowManager`*.

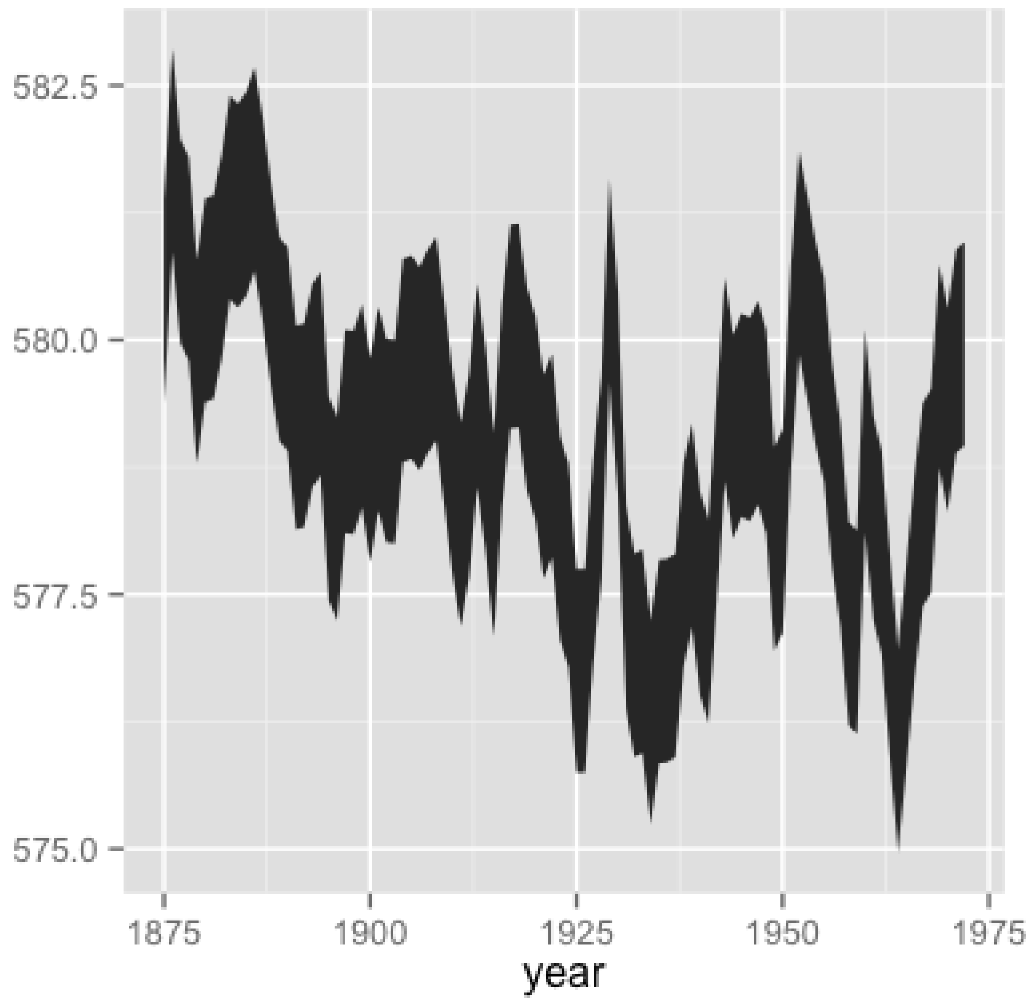

Fig. 4.1: Image credit: taken from the "ggplot2" documentation on 26 November 2013; reused here under the GNU General Public License, version 2.

- Limitation: This is a point of information for users and developers concerned with counterpoint. The framework currently offers no way to sensitively process voice crossing in contrapuntal modules ("interval n-grams"). "Higher" and "lower" voices are consistently presented in score order. We have planned for several ways to deal with this situation, but the developer assigned to the task is a busy doctoral student and a novice programmer, so they have not been fully implemented yet.

4.4 Tutorial: Make a new Workflow

Once you understand our framework's architecture (explained in *Design Principles*), you can start to design a new workflow to ask your own queries.

4.4.1 Develop a Question

Pretend you are me. I want to describe what distinguishes the melodic styles of two composers. I have already chosen the composers and the pieces I will use to compare them, trying to make the test sets as similar as possible except for the different composers. I want to use the vis framework, and I want to be as lazy as possible, so I will try to avoid adding new analyzers.

For my preliminary investigation, I will consider only patterns of melodic motion, since all required indexers and experimenters are already included with the vis framework. The *`NGramIndexer`* provides vis with pattern-finding functionality, so to run my query I must consider two questions: (1) what does the `NGramIndexer` need in order to find melodic patterns? and (2) how shall I aggregate the melodic patterns across pieces?

After the preliminary investigation, I would make my query more useful by using the "horizontal" and "vertical" functionality of the `NGramIndexer` to coordinate disparate musical elements that make up melodic identity. Writing a new `Indexer` to help combine melodic intervals with the duration of the note preceding the interval would be relatively easy, since music21 knows the duration of every note. A more subtle, but possibly more informative, query would combine melodic intervals with the scale degree of the preceding note. This is a much more complicated query, since it would require an indexer to find the key at a particular moment (an extremely complicated question) and an indexer that knows the scale degree of a note.

4.4.2 What Does the NGramIndexer Require?

I start by reading and understanding the documentation for the *`NGramIndexer`*. This indexer's power means it can be difficult to use in subtle and unexpected ways. For this simple preliminary investigation, we need only provide the melodic intervals of every part in an `IndexedPiece`. The melodic intervals will be the "vertical" events; there will be no "horizontal" events. We can change the "mark singles" and "continuer" settings any time as we please. We will probably want to try many different pattern lengths by changing the "n" setting. If we do not wish our melodic patterns to include rests, we can set "terminator" to `[u'Rest']`.

Thus the only information `NGramIndexer` requires from another analyzer is the melodic intervals, produced by `HorizontalIntervalIndexer`, which will confusingly be the "vertical" event. As specified in its documentation, the `HorizontalIntervalIndexer` requires the output of the *`NoteRestIndexer`*, which operates directly on the music21 `Score`.

The first part of our query looks like this:

```python
from vis.analyzers.indexers import noterest, interval, ngram
from vis.models.indexed_piece import IndexedPiece

# prepare inputs and output-collectors
pathnames = [list_of_pathnames_here]
ind_ps = [IndexedPiece(x) for x in pathnames]
```

```
7  interval_settings = {'quality': True}
8  ngram_settings = {'vertical': 0, 'n': 3}  # change 'n' as required
9  ngram_results = []
10
11 # prepare for and run the NGramIndexer
12 for piece in ind_ps:
13     intervals = piece.get_data([noterest.NoteRestIndexer, interval.HorizontalIntervalIndexer], inter
14     for part in intervals:
15         ngram_results.append(piece.get_data([ngram.NGramIndexer], ngram_settings, [part])
```

After the imports, we start by making a list of all the pathnames to use in this query, then use a Python list comprehension to make a list of `IndexedPiece` objcects for each file. We make the settings dictionaries to use for the interval then n-gram indexers on lines 7 and 8, but note we have not included all possible settings. The empty `ngram_results` list will store results from the `NGramIndexer`.

The loop started on line 12 is a little confusing: why not use an `AggregatedPieces` object to run the `NGramIndexer` on all pieces with a single call to `get_data()`? The reason is the inner loop, started on line 14: if we run the `NGramIndexer` on an `IndexedPiece` once, we can only index a single part, but we want results from all parts. This is the special burden of using the `NGramIndexer`, which is flexible but not (yet) intelligent. In order to index the melodic intervals in every part using the `get_data()` call on line 15, we must add the nested loops.

4.4.3 How Shall I Aggregate Results?

For this analysis, I will simply count the number of occurrences of each harmonic interval pattern, which is called the "frequency." It makes sense to calculate each piece separately, then combine the results across pieces. We'll use the *`FrequencyExperimenter`* and *`ColumnAggregator`* experimenters for these tasks. The `FrequencyExperimenter` counts the number of occurrences of every unique token in another index into a `pandas.Series`, and the `ColumnAggregator` combines results across a list of `Series` or a `DataFrame` (which it treats as a list of `Series`) into a single `Series`.

With these modifications, our program looks like this:

```
from vis.analyzers.indexers import noterest, interval, ngram
from vis.analyzers.experimenters import frequency, aggregator
from vis.models.indexed_piece import IndexedPiece
from vis.models.aggregated_pieces import AggregatedPieces
from pandas import DataFrame

# prepare inputs and output-collectors
pathnames = [list_of_pathnames_here]
ind_ps = [IndexedPiece(x) for x in pathnames]
interval_settings = {'quality': True}
ngram_settings = {'vertical': [0], 'n': 3}  # change 'n' as required
ngram_freqs = []

# prepare for and run the NGramIndexer
for piece in ind_ps:
    intervals = piece.get_data([noterest.NoteRestIndexer, interval.HorizontalIntervalIndexer], inter
    for part in intervals:
        ngram_freqs.append(piece.get_data([ngram.NGramIndexer, frequency.FrequencyExperimenter], ngra

# aggregate results of all pieces
agg_p = AggregatedPieces(ind_ps)
result = agg_p.get_data([aggregator.ColumnAggregator], [], {}, ngram_freqs)
result = DataFrame({'Frequencies': result})
```

The first thing to note is that I modified the loop from the previous step by adding the `FrequencyExperimenter` to the `get_data()` call on line 18 that uses the `NGramIndexer`. As you can see, the aggregation step is actually the easiest; it simply requires we create an `AggregatedPieces` object and call its `get_data()` method with the appropriate input, which is the frequency data we collected in the loop.

On line 22, `result` holds a `Series` with all the information we need! To export your data to one of the supported formats (CSV, Excel, etc.) you must create a `DataFrame` and use one of the methods described in the pandas documentation. The code on line 23 "converts" `result` into a `DataFrame` by giving the `Series` to the `DataFrame` constructor in a dictionary. The key is the name of the column, which you can change to any value valid as a Python dictionary key. Since the `Series` holds the frequencies of melodic interval patterns, it makes sense to call the column `'Frequencies'` in this case. You may also wish to sort the results by running `result.sort()` before you "convert" to a `DataFrame`. You can sort in descending order (with the most common events at the top) with `result.sort(ascending=False)`.

4.5 Tutorial: Use the WorkflowManager

The script developed in *Tutorial: Make a new Workflow* is suitable for users comfortable with an interactive Python shell. Application developers making a graphical interface—whether on the Web or in a desktop application—can take advantage of a further layer of abstraction offered by our *`WorkflowManager`*. Since developers often prefer to separate their user interface code from any of the so-called "business logic," the `WorkflowManager` provides the means by which to connect the "dumb" user interface with the highly-abstracted vis framework. You can think of the `WorkflowManager` as the true back-end component of your application, and you should expect to rewrite it with every application you develop.

The `WorkflowManager`'s documentation describes its functionality:

class `vis.workflow.`**`WorkflowManager`**(*pathnames*)

Parameters **`pathnames`** (list or tuple of basestring or *`IndexedPiece`*) – A list of pathnames.

The `WorkflowManager` automates several common music analysis patterns for counterpoint. Use the `WorkflowManager` with these four tasks:

- `load()`, to import pieces from symbolic data formats.
- `run()`, to perform a pre-defined analysis.
- `output()`, to output a visualization of the analysis results.
- `export()`, to output a text-based version of the analysis results.

Before you analyze, you may wish to use these methods:

- `metadata()`, to get or set the metadata of a specific `IndexedPiece` managed by this `WorkflowManager`.
- `settings()`, to get or set a setting related to analysis (for example, whether to display the quality of intervals).

You may also treat a `WorkflowManager` as a container:

```
>>> wm = WorkflowManager(['piece1.mxl', 'piece2.krn'])
>>> len(wm)
2
>>> ip = wm[1]
>>> type(ip)
<class 'vis.models.indexed_piece.IndexedPiece'>
```

4.5.1 Port a Query into the WorkflowManager

If I want to port the *Tutorial: Make a new Workflow* query to the `WorkflowManager`, I need to fit its functionality into the existing methods. The `load()`, `output()`, and `export()` methods are all related to preparing `IndexedPiece` objects for analysis and saving or outputting results. Since my query requires no special treatment in these areas, I will not modify those methods, and all of my changes will be in the `run()` method.

Since my new program only requires one query, I can make a very simple `run()` method and remove the other hidden methods (`_intervs()`, `_interval_ngrams()`, `_variable_part_modules()`, `_two_part_modules()`, and `_all_part_modules()`). Of course, you may wish to use those methods for inspiration when you build your own queries. When I add my new query's logic to the `run()` method, I get this:

```
1  def run(self):
2      ngram_settings = {'vertical': [0], 'n': self.settigns(None, 'n')}
3      ngram_freqs = []
4  
5      for i, piece in enumerate(self._data):
6          interval_settings = {'quality': self.settings(i, 'interval quality')}
7          intervals = piece.get_data( \
8              [noterest.NoteRestIndexer, interval.HorizontalIntervalIndexer], \
9              interval_settings)
10         for part in intervals:
11             ngram_freqs.append( \
12                 piece.get_data([ngram.NGramIndexer, frequency.FrequencyExperimenter], \
13                                ngram_settings, \
14                                [part]))
15 
16     agg_p = AggregatedPieces(ind_ps)
17     self._result = agg_p.get_data([aggregator.ColumnAggregator], [], {}, ngram_freqs)
```

I made the following changes:

- Remove the `instruction` parameter from `run()`, since there is only one experiment.
- Use the `import` statements at the top of the file.
- Use `self._data` rather than building my own list of `IndexedPiece` objects (in `enumerate()` on line 5).
- Set `interval_settings` per-piece, and use the value from built-in `WorkflowManager` settings.
- Set `n` from the built-in `WorkflowManager` settings.

I could also use the `WorkflowManager.settings()` method to get other settings by piece or shared across all pieces, like `'simple intervals'`, which tells the `HorizontalIntervalIndexer` whether to display all intervals as their single-octave equivalents.

To run the same analysis as in *Tutorial: Make a new Workflow*, use the `WorkflowManager` like this:

```
from vis.workflow import WorkflowManager

pathnames = [list_of_pathnames]
work = WorkflowManager(pathnames)
work.load('pieces')
for i in xrange(len(work)):
    work.settings(i, 'quality', True)
work.run()
work.export('CSV', 'output_filename.csv')
```

This script actually does more than the program in *Tutorial: Make a new Workflow* because `export()` "converts" the results to a `DataFrame`, sorts, and outputs the results.

CHAPTER 5

API Specification

5.1 vis

5.1.1 analyzers Package

`analyzers` Package

Indexers produce an "index" of a symbolic musical score. Each index provides one type of data, and each event in can be attached to a particular moment in the original score. Some indexers produce their index directly from the score, like the `NoteRestIndexer`, which describes pitches. Others create new information by analyzing another index, like the `IntervalIndexer`, which describes harmonic intervals between two-part combinations in the score, or the `FilterByRepeatIndexer`, which removes consecutive identical events, leaving only the first.

Analysis modules that subclass *`Experimenter`*, by contrast, produce results that cannot be attached to a moment in a score.

Indexers work only on single *`IndexedPiece`* instances. To analyze many *`IndexedPiece`* objects together, use an experimenter with an *`AggregatedPieces`* object.

`experimenter` Module

This module outlines the Experimenter base class, which helps with transforming time-attached analytic information to other types.

class `vis.analyzers.experimenter.`**`Experimenter`**`(`*index*, *settings=None*`)`

Bases: `object`

Run an experiment on an IndexedPiece.

Use the "Experimenter.required_indices" attribute to know which Indexer subclasses should be provided to this Experimenter's constructor. If the list is None or [], use the "Experimenter.required_experiments" attribute to know which Experimenter should be provided to the constructor.

`default_settings` **= None**

`possible_settings` **= []**

`run``()`

Run an experiment on a piece.

Returns The result of the experiment. Data is stored uniquely depending on the Experiment.

Return type pandas.Series or pandas.DataFrame

`indexer` Module

The controllers that deal with indexing data from music21 Score objects.

class `vis.analyzers.indexer.`**`Indexer`**(*score*, *settings=None*)

Bases: `object`

An object that manages creating an index of a piece, or part of a piece, based on one feature.

Use the *`required_score_type`* attribute to know what type of object is required in `__init__()`.

The name of the indexer, as stored in the `DataFrame` it returns, is the module name and class name. For example, the name of the *`IntervalIndexer`* is `'interval.IntervalIndexer'`.

Caution: This module underwent significant changes for release 2.0.0. In particular, the constructor's `score` argument and the *`run()`* method's return type have changed.

`default_settings` = {}

Described in the *`TemplateIndexer`*.

`make_return`(*labels*, *indices*)

Prepare a properly-formatted `DataFrame` as should be returned by any *`Indexer`* subclass. We intend for this to be called by *`Indexer`* subclasses only.

The index of a label in `labels` should be the same as the index of the `Series` to which it corresponds in `indices`. For example, if `indices[12]` is the tuba part, then `labels[12]` might say `'Tuba'`.

Parameters

- **`labels`** (*list of basestring*) – Indices of the parts or the part combinations, or another descriptive label as described in the indexer subclass documentation.
- **`indices`** (list of `pandas.Series`.) – The results of the indexer.

Returns A `DataFrame` with the appropriate `MultiIndex` required by the *`Indexer.run()`* method signature.

Return type `pandas.DataFrame`

Raises `IndexError` if the number of labels and indices does not match.

`possible_settings` = {}

Described in the *`TemplateIndexer`*.

`required_score_type` = None

Described in the *`TemplateIndexer`*.

`run`()

Make a new index of the piece.

Returns The new indices. Refer to the section below.

Return type `pandas.DataFrame`

About Return Values:

Every indexer must return a `DataFrame` with a special kind of `MultiIndex` that helps organize data across multiple indexers. Programmers making a new indexer should follow the instructions in the `TemplateIndexer` *`run()`* method to ensure this happens properly.

Indexers return a `DataFrame` where the columns are indexed on two levels: the first level is a string with the name of the indexer, and the second level is a string with the index of the part, the indices of the parts in a combination, or some other value as specified by the indexer.

This allows, for example:

```
>>> the_score = music21.converter.parse('sibelius_5-i.mei')
>>> the_score.parts[5]
(the first clarinet Part)
>>> the_notes = NoteRestIndexer(the_score).run()
>>> the_notes['noterest.NoteRestIndexer']['5']
(the first clarinet Series)
>>> the_intervals = IntervalIndexer(the_notes).run()
>>> the_intervals['interval.IntervalIndexer']['5,6']
(Series with vertical intervals between first and second clarinet)
```

This is more useful when you have a larger `DataFrame` with the results of multiple indexers. Refer to `Indexer.combine_results()` to see how that works.

```
>>> some_results = Indexer.combine_results([the_notes, the_intervals])
>>> some_results['noterest.NoteRestIndexer']['5']
(the first clarinet Series)
>>> some_results['interval.IntervalIndexer']['5,6']
(Series with vertical intervals between first and second clarinet)
>>> some_results.to_hdf('brahms3.h5', 'table')
```

After the call to `to_hdf()`, your results are stored in the 'brahms3.h5' file. When you load them (very quickly!) with the `read_hdf()` method, the `DataFrame` returns exactly as it was.

Note: In release 1.0.0, it was sometimes acceptable to use undocumented return values; from release 1.1.0, this is no longer necessary, and you should avoid it. In a future release, the `IndexedPiece` class will depend on indexers following these rules.

`vis.analyzers.indexer.`**`mpi_unique_offsets`**`(`*streams*`)`

For a set of `Stream` objects, find the offsets at which events begin. Used by *`stream_indexer()`*.

Parameters **`streams`** (list of `music21.stream.Stream`) – A list of `Stream` objects in which to find the offsets where events begin.

Returns A list of floating-point numbers representing offsets at which a new event begins in any of the `Stream` objects. Offsets are sorted from lowest to highest (start to end).

Return type list of float

`vis.analyzers.indexer.`**`series_indexer`**`(`*pipe_index*, *parts*, *indexer_func*`)`

Perform the indexation of a part or part combination. This is a module-level function designed to ease implementation of multiprocessing.

If your *`Indexer`* has settings, use the `indexer_func()` to adjust for them.

Parameters

- **`pipe_index`** (*object*) – An identifier value for use by the caller. This is returned unchanged, so a caller may use the `pipe_index` as a tag with which to keep track of jobs.
- **`parts`** (list of `pandas.Series`) – A list of at least one `Series` object. Every new event, or change of simlutaneity, will appear in the outputted index. Therefore, the new index will contain at least as many events as the inputted `Series` with the most events. This is not a `DataFrame`, since each part will likely have different offsets.
- **`indexer_func`** (*function*) – This function transforms found events into a unicode object.

Returns The `pipe_index` argument and the new index. The new index is a `pandas.Series` where every element is a unicode string. The `Index` of the `Series` corresponds to the `quarterLength` offset of the event in the inputted `Stream`.

Return type 2-tuple of object and `pandas.Series`

Raises `ValueError` if there are multiple events at an offset in any of the inputted `Series`.

`vis.analyzers.indexer.`**`stream_indexer`**(*pipe_index*, *parts*, *indexer_func*, *types=None*)

Perform the indexation of a `Part` or `Part` combination. This is a module-level function designed to ease implementation of multiprocessing.

If your *`Indexer`* subclass has settings, use the `indexer_func()` to adjust for them.

If an offset has multiple events of the correct type, only the "first" discovered results will be included in the output. This may produce misleading results when, for example, a double-stop was imported as two `Note` objects in the same `Part`, rather than as a `Chord`.

Parameters

- **`pipe_index`** (*object*) – An identifier value for use by the caller. This is returned unchanged, so a caller may use the `pipe_index` as a tag with which to keep track of jobs.
- **`parts`** (list of `music21.stream.Stream`) – A list of at least one `Stream` object. Every new event or change of simlutaneity will appear in the outputted index. Therefore, the new index will contain at least as many events as the inputted `Part` with the most events.
- **`indexer_func`** (*function*) – This function transforms found events into a unicode object.
- **`types`** (*list of type*) – Only objects of a type in this list will be passed to the *`indexer_func()`* for inclusion in the resulting index.

Returns The `pipe_index` argument and the new index. The new index is a `pandas.Series` where every element is a unicode string. The `Index` of the `Series` corresponds to the `quarterLength` offset of the event in the inputted `Stream`.

Return type 2-tuple of object and `pandas.Series`

Subpackages

experimenters Package

`experimenters` Package

`aggregator` Module Aggregating experimenters.

class `vis.analyzers.experimenters.aggregator.`**`ColumnAggregator`**(*index*, *settings=None*)

Bases: *`vis.analyzers.experimenter.Experimenter`*

(Arguments for the constructor are listed below).

Experiment that aggregates data from columns of a `DataFrame`, or a list of `DataFrame` objects, by summing each row. Values from columns named `'all'` will not be included in the aggregated results. You may provide a `'column'` setting to guide the experimenter to include only certain results.

Example 1

Inputting single `DataFrame` like this:

Index	piece_1	piece_2
M3	12	24
m3	NaN	36
P5	3	9

Yields this `DataFrame`:

Index	'aggregator.ColumnAggregator'
M3	36
m3	36
P5	12

Example 2

Inputting two `DataFrame` objects is similar.

Index	piece_1
M3	12
P5	3

Index	piece_2
M3	24
m3	36
P5	9

The result is the same `DataFrame`:

Index	'aggregator.ColumnAggregator'
M3	36
m3	36
P5	12

Example 3

You may also give a `DataFrame` (or a list of `DataFrame` objects) that have a `pandas.MultiIndex` as produced by subclasses of *`Indexer`*. In this case, use the `'column'` setting to indicate which indexer's results you wish to aggregate.

Index	'frequency.FrequencyExperimenter'		'feelings.FeelingsExperimenter'	
	'0,1'	'1,2'	'Christopher'	'Alex'
M3	12	24	'delight'	'exuberance'
m3	NaN	36	'sheer joy'	'nonchalance'
P5	3	9	'emptiness'	'serenity'

If `'column'` is `'frequency.FrequencyExperimenter'`, yet again you will have this `DataFrame`:

Index	'aggregator.ColumnAggregator'
M3	36
m3	36
P5	12

`default_settings` = {'column': None}

`possible_settings` = ['column']

Parameters **`'column'`** (*str*) – The column name to use for aggregation. The default is `None`, which aggregates across all columns. If you set this to `'all'`, it will override the default behaviour of not including columns called `'all'`.

`run`()

Run the *`ColumnAggregator`* experiment.

Returns A `Series` with an index that is the combination of all indices of the provided pandas objects, and the value is the sum of all values in the pandas objects.

Return type `pandas.Series`

`frequency` Module Experimenters that deal with the frequencies (number of occurrences) of events.

class `vis.analyzers.experimenters.frequency.`**`FrequencyExperimenter`**(*index*, *settings=None*)

Bases: *`vis.analyzers.experimenter.Experimenter`*

Calculate the number of occurrences of objects in an index.

Use the `'column'` setting to choose only the results of one previous analyzer. For example, if you wanted to calculate the frequency of vertical intervals, you would specify `'interval.IntervalIndexer'`. This would avoid counting, for example, the horizontal intervals if they were also present.

`default_settings` = {'column': None}

`possible_settings` = ['column']

Parameters **`'column'`** (*str*) – The column name to use for counting frequency. The default is `None`, which counts all columns. Use this to count only the frequency of one previous analyzer.

`run`()

Run the *`FrequencyExperimenter`*.

Returns The result of the experiment. Data is stored such that column labels correspond to the part (combinations) totalled in the column, and row labels correspond to a type of the kind of objects found in the given index. Note that all columns are totalled in the "all" column, and that not every part combination will have every interval; in case an interval does not appear in a part combination, the value is `numpy.NaN`.

Return type list of `pandas.DataFrame`

`lilypond` Module Experimenters related to producing LilyPond-format output from the VIS Framework. Also refer to the *`vis.analyzers.indexers.lilypond`* module.

The *`LilyPondExperimenter`* uses the `outputlilypond` module to produce a LilyPond file corresponding to the score.

class `vis.analyzers.experimenters.lilypond.`**`AnnotateTheNoteExperimenter`**(*index*, *settings=None*)

Bases: *`vis.analyzers.experimenter.Experimenter`*

Make a new `Note` object with the input set to the `lily_markup` property, the `lily_invisible` property set to `True`, and everything else as a default `Note`.

`possible_settings` = ['column']

Use the `'column'` setting to determine which column of the `DataFrame` will be used as the annotations for the notes in the outputted list of `Series`.

`run`()

Make a new index of the piece.

Returns A list of the new indices. The index of each `Series` corresponds to the index of the `Part` used to generate it, in the order specified to the constructor. Each element in the `Series` is a `basestring`.

Return type list of `pandas.Series`

class `vis.analyzers.experimenters.lilypond.`**`LilyPondExperimenter`**(*index*, *settings=None*)

Bases: *`vis.analyzers.experimenter.Experimenter`*

Use the `outputlilypond` module to produce the LilyPond file that should produce a score of the input.

Note: Perhaps contrary to expectation, you must provide a `music21.stream.Score` to the *`LilyPondExperimenter`*, and any part with annotations belong in the settings.

`default_settings` = **{'annotation_part': None, 'run_lilypond': False, 'output_pathname': None}**

`possible_settings` = **['run_lilypond', 'output_pathname', 'annotation part']**

Possible settings for the *`LilyPondExperimenter`* include:

Parameters

- **`'run_lilypond'`** (*boolean*) – Whether to run LilyPond; if `False` or omitted, simply produce the input file LilyPond requires.
- **`'output_pathname'`** (*basestring*) – Pathname for the resulting LilyPond output file. If `'run_lilypond'` is `True`, you must include this setting. If `'run_lilypond'` is `False` and you do not provide `'output_pathname'` then the output file is returned by *`run()`* as a `unicode`.
- **`'annotation_part'`** (`music21.stream.Part` or list of `Part`) – A `Part` or list of `Part` objects with annotation instructions for `outputlilypond`. This `Part` will be appended as last in the `Score`.

`required_score_type` = **'stream.Score'**

This attribute allows `IndexedPiece` to automatically import and provide the `Score` for *`LilyPondExperimenter`*. Otherwise you would have to do this manually.

`run`()

Make a string with the LilyPond representation of each score. Run LilyPond, if we're supposed to.

Returns A string holding the LilyPond-format representation of the score and its annotation parts.

Return type basestring

class `vis.analyzers.experimenters.lilypond.`**`PartNotesExperimenter`**(*score*, *settings=None*)

Bases: *`vis.analyzers.experimenter.Experimenter`*

From a `Series` full of `Note` objects, craft a `music21.stream.Part`. The offset of each `Note` in the output matches its index in the input `Series`, and each `duration` property is set to match.

To print a "name" along with the first item in a part, for example to indicate to which part or part combinations the annotations belong, use the optional `part_names` setting.

`default_settings` = **{}**

`possible_settings` = **['part_names']**

Parameters **`part_names`** (*list of basestring*) – Names for the annotation parts, in order. If there are more part names than parts, extra names will be ignored. If there are fewer part names than parts, some parts will not be named.

`required_score_type`

alias of `Series`

`run`()

Make a new index of the piece.

Returns A list of the new indices. The index of each `Part` corresponds to the index of the `Series` used to generate it, in the order specified to the constructor. Each element in the `Part` is a `Note`.

Return type list of `music21.stream.Part`

`vis.analyzers.experimenters.lilypond.`**`annotate_the_note`**(*obj*)

Used by *`AnnotateTheNoteExperimenter`* to make a `Note` object with the annotation passed in. Take note (hahaha): the `lily_invisible` property is set to `True`!

Parameters **`obj`** (*basestring*) – A string to put as the `lily_markup` property of a new `Note`.

Returns An annotated note.

Return type `music21.note.Note`

`template` **Module** Template for writing a new experimenter. Use this class to help write a new :class‘Experimenter‘ subclass. The *`TemplateExperimenter`* does nothing, and should only be used by programmers.

class `vis.analyzers.experimenters.template.`**`TemplateExperimenter`**(*index*, *settings=None*)

Bases: *`vis.analyzers.experimenter.Experimenter`*

Template for an `Experimenter` subclass.

`default_settings` **= {}**

The default values for settings named in *`possible_settings`*. If a setting doesn't have a value in this constant, then it must be specified to the constructor at runtime, or the constructor should raise a `RuntimeException`.

`possible_settings` **= ['fake_setting']**

This is a list of basestrings that are the names of the settings used in this experimenter. Specify the types and reasons for each setting as though it were an argument list, like this:

Parameters **`'fake_setting'`** (*boolean*) – This is a fake setting.

`run`()

Run an experiment on a piece.

Returns The result of the experiment. Each experiment should describe its data storage.

Return type `pandas.Series` or `pandas.DataFrame`

indexers Package

`indexers` Package

`interval` **Module** Index intervals. Use the *`IntervalIndexer`* to find vertical (harmonic) intervals between two parts. Use the *`HorizontalIntervalIndexer`* to find horizontal (melodic) intervals in the same part.

class `vis.analyzers.indexers.interval.`**`HorizontalIntervalIndexer`**(*score*, *settings=None*)

Bases: *`vis.analyzers.indexers.interval.IntervalIndexer`*

Use `music21.interval.Interval` to create an index of the horizontal (melodic) intervals in a single part.

You should provide the result of `NoteRestIndexer`.

`default_settings` **= {'horiz_attach_later': False}**

`possible_settings` **= ['horiz_attach_later']**

This setting applies to the *`HorizontalIntervalIndexer`* *in addition to* the settings available from the *`IntervalIndexer`*.

Parameters **'horiz_attach_later'** (*boolean*) – If `True`, the offset for a horizontal interval is the offset of the later note in the interval. The default is `False`, which gives horizontal intervals the offset of the first note in the interval.

run()

Make a new index of the piece.

Returns The new indices. Refer to the example below.

Return type `pandas.DataFrame`

Example:

```
>>> the_score = music21.converter.parse('sibelius_5-i.mei')
>>> the_score.parts[5]
(the first clarinet Part)
>>> the_notes = NoteRestIndexer(the_score).run()
>>> the_notes['noterest.NoteRestIndexer']['5']
(the first clarinet Series)
>>> the_intervals = HorizontalIntervalIndexer(the_notes).run()
>>> the_intervals['interval.HorizontalIntervalIndexer']['5']
(Series with melodic intervals of the first clarinet)
```

class vis.analyzers.indexers.interval.**IntervalIndexer**(*score*, *settings=None*)

Bases: *vis.analyzers.indexer.Indexer*

Use `music21.interval.Interval` to create an index of the vertical (harmonic) intervals between two-part combinations.

You should provide the result of the *NoteRestIndexer*. However, to increase your flexibility, the constructor requires only a list of `Series`. You may also provide a `DataFrame` exactly as outputted by the `NoteRestIndexer`.

default_settings = {u'simple or compound': u'compound', u'quality': False}

A dict of default settings for the *IntervalIndexer*.

possible_settings = [u'simple or compound', u'quality']

A list of possible settings for the *IntervalIndexer*.

Parameters

- **u'simple or compound'** (*unicode*) – Whether intervals should be represented in their single-octave form (either `u'simple'` or `u'compound'`).
- **u'quality'** (*boolean*) – Whether to display an interval's quality.

required_score_type = 'pandas.Series'

run()

Make a new index of the piece.

Returns A `DataFrame` of the new indices. The columns have a `MultiIndex`; refer to the example below for more details.

Return type `pandas.DataFrame`

Example:

```
>>> the_score = music21.converter.parse('sibelius_5-i.mei')
>>> the_score.parts[5]
(the first clarinet Part)
>>> the_notes = NoteRestIndexer(the_score).run()
>>> the_notes['noterest.NoteRestIndexer']['5']
(the first clarinet Series)
```

```
>>> the_intervals = IntervalIndexer(the_notes).run()
>>> the_intervals['interval.IntervalIndexer']['5,6']
(Series with vertical intervals between first and second clarinet)
```

`vis.analyzers.indexers.interval.`**`indexer_nq_comp`**(*ecks*)
Used internally by the `IntervalIndexer` and `HorizontalIntervalIndexer`.

Call `real_indexer()` with settings to print compound intervals without quality.

`vis.analyzers.indexers.interval.`**`indexer_nq_simple`**(*ecks*)
Used internally by the `IntervalIndexer` and `HorizontalIntervalIndexer`.

Call `real_indexer()` with settings to print simple intervals without quality.

`vis.analyzers.indexers.interval.`**`indexer_qual_comp`**(*ecks*)
Used internally by the `IntervalIndexer` and `HorizontalIntervalIndexer`.

Call `real_indexer()` with settings to print compound intervals with quality.

`vis.analyzers.indexers.interval.`**`indexer_qual_simple`**(*ecks*)
Used internally by the `IntervalIndexer` and `HorizontalIntervalIndexer`.

Call `real_indexer()` with settings to print simple intervals with quality.

`vis.analyzers.indexers.interval.`**`real_indexer`**(*simultaneity*, *simple*, *quality*)
Used internally by the `IntervalIndexer` and `HorizontalIntervalIndexer`.

Parameters

- **`simultaneity`** (*list of basestring*) – A two-item iterable with the note names for the higher and lower parts, respectively.
- **`simple`** (*boolean*) – Whether intervals should be reduced to their single-octave version.
- **`quality`** (*boolean*) – Whether the interval's quality should be prepended.

Returns `'Rest'` if one or more of the parts is `'Rest'`; otherwise, the interval between the parts.

Return type unicode string

`lilypond` **Module** Indexers related to producing LilyPond-format output from the VIS Framework. Also refer to the `vis.analyzers.experimenters.lilypond` module.

class `vis.analyzers.indexers.lilypond.`**`AnnotationIndexer`**(*score*, *settings=None*)
Bases: `vis.analyzers.indexer.Indexer`

From any other index, put `_\markup{""}` around it.

`required_score_type` **= 'pandas.Series'**

`run`()
Make a new index of the piece.

Returns A list of the new indices. The index of each `Series` corresponds to the index of the `Part` used to generate it, in the order specified to the constructor. Each element in the `Series` is a `basestring`.

Return type `pandas.DataFrame`

`vis.analyzers.indexers.lilypond.`**`annotation_func`**(*obj*)
Used by `AnnotationIndexer` to make a "markup" command for LilyPond scores.

Parameters **`obj`** (`pandas.Series` of `unicode`) – A single-element `Series` with the string to wrap in a "markup" command.

Returns The thing in a markup.

Return type `unicode`

`ngram` Module Indexer to find k-part any-object n-grams.

class `vis.analyzers.indexers.ngram.`**`NGramIndexer`**(*score*, *settings=None*)
Bases: *`vis.analyzers.indexer.Indexer`*

Indexer that finds k-part n-grams from other indices.

The indexer requires at least one "vertical" index, and supports "horizontal" indices that seem to "connect" instances in the vertical indices. Although we use "vertical" and "horizontal" to describe these index types, because the class is an abstraction of two-part interval n-grams, you can supply any information as either type of index. If you want one-part melodic n-grams for example, you should supply the relevant interval information as the "vertical" component.

There is no relationship between the number of index types, though there must be at least one "vertical" index.

The `'horizontal'` and `'vertical'` settings determine which columns of the `score` `DataFrame` are included in the n-gram output. They are added to the n-gram in the order specified, so if the `'vertical'` setting is `[('noterest.NoteRestIndexer', '1'), ('noterest.NoteRestIndexer', '0')]`, this will put the lower part (with index `'1'`) before the higher part (with index `'0'`). Note that both the indexer's name and the part-combination name must be included.

This is an example minimum `settings` dictionary for making interval 3-grams::

```
{'vertical': [('interval.IntervalIndexer', '0,1')],
 'horizontal': [('interval.HorizontalIntervalIndexer', '1')],
 'n': 3}
```

In the output, groups of "vertical" events are normally enclosed in brackets, while groups of "horizontal" events are enclosed in parentheses. For cases where there is only one index in a particular direction, you can avoid printing the brackets or parentheses by setting the `'mark singles'` setting to `False` (the default is `True`).

If you want n-grams to terminate when finding one or several particular values, you can specify this with the `'terminator'` setting.

To show that a horizontal event continues, we use `'_'` by default, but you can set this separately, for example to `'P1'` `'0'`, as seems appropriate. Note that the default `WorkflowManager` overrides this setting by dynamically adjusting for interval quality, and also offers a `'continuer'` setting of its own, which is passed to this indexer.

You can also use the *`NGramIndexer`* to collect "stacks" of single vertical events. If you provide indices of intervals above a lowest part, for example, these "stacks" become the figured bass signature of a single moment. Set `'n'` to `1` for this feature. Horizontal events are obviously ignored in this case.

`default_settings` **= {'horizontal': [], 'continuer': '_', 'mark_singles': True, 'terminator': []}**

`possible_settings` **= ['horizontal', 'vertical', 'n', 'mark_singles', 'terminator', 'continuer']**
A list of possible settings for the *`NGramIndexer`*.

Parameters

- **`'horizontal'`** (*list of (basestring, basestring) tuples*) – Selectors for the parts to consider as "horizontal."
- **`'vertical'`** (*list of (basestring, basestring) tuples*) – Selectors for the parts to consider as "vertical."
- **`'n'`** (*int*) – The number of "vertical" events per n-gram.

- **'mark_singles'** (*bool*) – Whether to use delimiters around a direction's events when there is only one event in that direction (e.g., the "horizontal" maps only the activity of a single voice). (You may also use `'mark singles'`).
- **'terminator'** (*list of basestring*) – Do not find an n-gram with a vertical item that contains any of these values.
- **'continuer'** (*basestring*) – When there is no "horizontal" event that corresponds to a vertical event, this is printed instead, to show that the previous "horizontal" event continues.

required_score_type = 'pandas.DataFrame'

run()

Make an index of k-part n-grams of anything.

Returns A single-column `DataFrame` with the new index.

noterest Module Index note and rest objects.

class `vis.analyzers.indexers.noterest.`**NoteRestIndexer**(*score*, *settings=None*)

Bases: *`vis.analyzers.indexer.Indexer`*

Index `Note` and `Rest` objects in a `Part`.

`Rest` objects become `'Rest'`, and `Note objects become the unicode-format version of their :attr:`~music21.note.Note.nameWithOctave` attribute.

required_score_type = 'stream.Part'

run()

Make a new index of the piece.

Returns A `DataFrame` of the new indices. The columns have a `MultiIndex`; refer to the example below for more details.

Return type `pandas.DataFrame`

Example:

```
>>> the_score = music21.converter.parse('sibelius_5-i.mei')
>>> the_score.parts[5]
(the first clarinet Part)
>>> the_notes = NoteRestIndexer(the_score).run()
>>> the_notes['noterest.NoteRestIndexer']['5']
(the first clarinet Series)
```

`vis.analyzers.indexers.noterest.`**indexer_func**(*obj*)

Used internally by *`NoteRestIndexer`*. Convert `Note` and `Rest` objects into a `unicode` string.

Parameters **obj** (iterable of `music21.note.Note` or `music21.note.Rest`) – An iterable (nominally a `Series`) with an object to convert. Only the first object in the iterable is processed.

Returns If the first object in the list is a `music21.note.Rest`, the string `u'Rest'`; otherwise the `nameWithOctave` attribute, which is the pitch class and octave of the `Note`.

Return type unicode

Examples:

```
>>> from music21 import note
>>> indexer_func([note.Note('C4')])
u'C4'
```

```
>>> indexer_func([note.Rest()])
u'Rest'
```

`offset` Module Indexers that modify the “offset” values (floats stored as the “index” of a `pandas.Series`), potentially adding repetitions of or removing pre-existing events, without modifying the events themselves.

class `vis.analyzers.indexers.offset.`**`FilterByOffsetIndexer`**(*score*, *settings=None*)
Bases: *`vis.analyzers.indexer.Indexer`*

Indexer that regularizes the “offset” values of observations from other indexers.

The Indexer regularizes observations from offsets spaced any, possibly irregular, quarterLength durations apart, so they are instead observed at regular intervals. This has two effects:

- events that do not begin at an observed offset will only be included in the output if no other event occurs before the next observed offset
- events that last for many observed offsets will be repeated for those offsets

Since elements’ durations are not recorded, the last observation in a Series will always be included in the results. If it does not start on an observed offset, it will be included as the next observed offset—again, whether or not this is true in the actual music. However, the last observation will only ever be counted once, even if a part ends before others in a piece with many parts. See the doctests for examples.

Examples. For all, the `quarterLength` is `1.0`.

When events in the input already appear at intervals of `quarterLength`, input and output are identical.

offset	0.0	1.0	2.0
input	a	b	c
output	a	b	c

When events in the input appear at intervals of `quarterLength`, but there are additional elements between the observed offsets, those additional elements are removed.

offset	0.0	0.5	1.0	2.0
input	a	A	b	c
output	a		b	c

offset	0.0	0.25	0.5	1.0	2.0
input	a	z	A	b	c
output	a			b	c

When events in the input appear at intervals of `quarterLength`, but not at every observed offset, the event from the previous offset is repeated.

offset	0.0	1.0	2.0
input	a		c
output	a	a	c

When events in the input appear at offsets other than those observed by the specified `quarterLength`, the “most recent” event will appear.

offset	0.0	0.25	0.5	1.0	2.0
input	a	z	A		c
output	a			A	c

When the final event does not appear at an observed offset, it will be included in the output at the next offset that would be observed, even if this offset does not appear in the score file to which the results correspond.

offset	0.0	1.0	1.5	2.0
input	`a`	`b`	`d`	
output	`a`	`b`		`d`

The behaviour in this last example can create a potentially misleading result for some analytic situations that consider metre. It avoids another potentially misleading situation where the final chord of a piece would appear to be dissonant because of a suspension. We chose to lose metric and rythmic precision, which would be more profitably analyzed with indexers built for that purpose. Consider this illustration, where the numbers correspond to scale degrees.

offset	410.0	411.0	411.5	412.0
in-S	2	1		
in-A	7	5		
in-T	4 ———		3	
in-B	5	1		
out-S	2	1		1
out-A	7	5		5
out-T	4	4		3
out-B	5	1		1

If we left out the note event appear in the `in-A` part at offset `411.5`, the piece would appear to end with a dissonant sonority!

`default_settings` **= {'method': 'ffill'}**

`possible_settings` **= ['quarterLength', 'method']**

A `list` of possible settings for the *`FilterByOffsetIndexer`*.

Parameters

- **`'quarterLength'`** (*float*) – The quarterLength duration between observations desired in the output. This value must not have more than three digits to the right of the decimal (i.e. 0.001 is the smallest possible value).
- **`'method'`** (*str or None*) – The value passed as the `method` kwarg to `reindex()`. The default is `'ffill'`, which fills in missing indices with the previous value. This is useful for vertical intervals, but not for horizontal, where you should use `None` instead.

`required_score_type` **= 'pandas.Series'**

The *`FilterByOffsetIndexer`* uses `pandas.Series` objects.

`run``()`

Regularize the observed offsets for the inputted Series.

Returns A `DataFrame` with offset-indexed values for all inputted parts. The pandas indices (holding music21 offsets) start at the first offset at which there is an event in any of the inputted parts. An offset appears every `quarterLength` until the final offset, which is either the last observation in the piece (if it is divisible by the `quarterLength`) or the next-highest value that is divisible by `quarterLength`.

Return type `pandas.DataFrame`

`repeat` **Module** Indexers that consider repetition in any way.

class `vis.analyzers.indexers.repeat.`**`FilterByRepeatIndexer`**`(`*score*, *settings=None*`)`

Bases: *`vis.analyzers.indexer.Indexer`*

If the same event occurs many times in a row, remove all occurrences but the one with the lowest `offset` value (i.e., the "first" event).

Because of how a `DataFrame`'s index works, many of the events that would have been filtered will instead be replaced with `numpy.NaN`. Please be careful that the behaviour of this indexer matches your expectations.

`required_score_type` = **'pandas.Series'**

`run``()`

Make a new index of the piece, removing any event that is identical to the preceding.

Returns A `DataFrame` of the new indices.

Return type `pandas.DataFrame`

`template` **Module** Template for writing a new indexer. Use this class to help write a new :class'Indexer' subclass. The *`TemplateIndexer`* does nothing, and should only be used by programmers.

Note: Follow these instructions to write a new `Indexer` subclass:

1. Replace my name with yours in the "codeauthor" directive above.
2. Change the "Filename" and "Purpose" on lines 7 and 8.
3. Modify the "Copyright" on line 10 *or* add an additional copyright line immediately below.
4. Remove the `# pylint:  disable=W0613` comment just before *`indexer_func()`*.
5. Rename the class.
6. Adjust `required_score_type`.
7. Add settings to `possible_settings` and `default_settings`, as required.
8. Rewrite the documentation for `__init__()`.
9. Rewrite the documentation for *`run()`*.
10. Rewrite the documentation for *`indexer_func()`*.
11. Write all relevant tests for `__init__()`, *`run()`*, and *`indexer_func()`*.
12. Follow the instructions in `__init__()` to write that method.
13. Follow the instructions in *`run()`* to write that method.
14. Write a new *`indexer_func()`*.
15. Ensure your tests pass, adding additional ones as required.
16. Finally, run `pylint` with the VIS style rules.

---

class `vis.analyzers.indexers.template.`**`TemplateIndexer`**`(`*score*, *settings=None*`)`

Bases: *`vis.analyzers.indexer.Indexer`*

Template for an `Indexer` subclass.

`default_settings` **= {}**

The default values for settings named in *`possible_settings`*. If a setting doesn't have a value in this constant, then it must be specified to the constructor at runtime, or the constructor should raise a `RuntimeException`.

`possible_settings` **= [u'fake_setting']**

This is a list of basestrings that are the names of the settings used in this indexer. Specify the types and reasons for each setting as though it were an argument list, like this:

Parameters **`'fake_setting'`** (*boolean*) – This is the description of a fake setting.

`required_score_type` = **'stream.Part'**

Depending on how this indexer works, you must provide a `DataFrame`, a `Score`, or list of `Part` or `Series` objects. Only choose `Part` or `Series` if the input will always have single-integer part combinations (i.e., there are no combinations—it will be each part independently).

`run``()`

Make a new index of the piece.

Returns The new indices. Refer to the note below.

Return type `pandas.DataFrame` or list of `pandas.Series`

Important: Please be sure you read and understand the rules about return values in the full documentation for *`run()`* and *`make_return()`*.

`vis.analyzers.indexers.template.`**`indexer_func`**`(`*obj*`)`

The function that indexes.

Parameters **`obj`** (list of objects of the types stored in `TemplateIndexer._types`) – The simultaneous event(s) to use when creating this index. (For indexers using a `Score`).

or

Parameters **`obj`** (`pandas.Series` of unicode strings) – The simultaneous event(s) to use when creating this index. (For indexers using a `Series`).

Returns The value to store for this index at this offset.

Return type `unicode`

5.1.2 models Package

`models` Package

`aggregated_pieces` Module

The model representing data from multiple *`IndexedPiece`* instances.

class `vis.models.aggregated_pieces.`**`AggregatedPieces`**`(`*pieces=None*`)`

Bases: `object`

Hold data from multiple *`IndexedPiece`* instances.

class **`Metadata`**

Bases: `object`

Used internally by *`AggregatedPieces`* ... at least for now.

Hold aggregated metadata about the IndexedPieces in an AggregatedPiece. Every list has no duplicate entries.

- composers: list of all the composers in the IndexedPieces
- dates: list of all the dates in the IndexedPieces
- date_range: 2-tuple with the earliest and latest dates in the IndexedPieces
- titles: list of all the titles in the IndexedPieces
- locales: list of all the locales in the IndexedPieces
- pathnames: list of all the pathnames in the IndexedPieces

composers

date_range

dates

locales

pathnames

titles

AggregatedPieces.**get_data**(*independent_analyzers*, *aggregated_experiments*, *settings=None*, *data=None*)

Get the results of an Experimenter run on all the IndexedPiece objects. You must specify all indexers and experimenters to be run to get the results you want.

The same settings dict will be given to all experiments and indexers.

If you want the results from all IndexedPiece objects separately, provide an empty list as the aggregated_experiments argument.

Either the first analyzer in independent_analyzers should use a music21.stream.Score or you must provide an argument for data that is the output from a previous call to this instance's *get_data()* method.

Examples

Run analyzer A then B on each piece individually, then provide a list of those results to Experimenter C then D::

```
>>> pieces.get_data([A, B], [C, D])
```

Run analyzer A then B on each piece individually, then return a list of those results::

```
>>> pieces.get_data([A, B])
```

Run experimenter A then B on the results of a previous *get_data()* call::

```
>>> piece.get_data([], [C, D], data=previous_results)
```

Note: The analyzers in the independent_analyzers argument are run with *get_data()* from the IndexedPiece objects themselves. Thus any exceptions raised there may also be raised here.

Parameters

- **independent_analyzers** (*list of types*) – The analyzers to run on each piece before aggregation, in the order you want to run them. For no independent analyzers, use [] or None.
- **aggregated_experiments** (*list of types*) – The Experimenters to run on aggregated data of all pieces, in the order you want to run them.
- **settings** (*dict*) – Settings to be used with the analyzers.
- **data** (pandas.DataFrame or list of DataFrame) – Input data for the first analyzer to run. If this argument is not None, you must provide the output from a previous call to *get_data()* of this instance.

Returns Either one pandas.DataFrame with all experimental results or a list of DataFrame objects, each with the experimental results for one piece.

Raises TypeError if an analyzer is invalid or cannot be found.

`AggregatedPieces.`**`metadata`**(*field*)

Get a metadatum about the IndexedPieces stored in this AggregatedPieces.

If only some of the stored IndexedPieces have had their metadata initialized, this method returns incompelete metadata. Missing data will be represented as `None` in the list, but it will not appear in `date_range` unless there are no dates. If you need full metadata, we recommend running an Indexer that requires a `Score` object on all the IndexedPieces (like *`vis.analyzers.indexers.noterest.NoteRestIndexer`*).

Valid fields are:

- `'composers`: list of all the composers in the IndexedPieces
- `'dates`: list of all the dates in the IndexedPieces
- `'date_range`: 2-tuple with the earliest and latest dates in the IndexedPieces
- `'titles`: list of all the titles in the IndexedPieces
- `'locales`: list of all the locales in the IndexedPieces
- `'pathnames`: list of all the pathnames in the IndexedPieces

Parameters **`field`** (`basestring`) – The name of the field to be accessed or modified.

Returns The value of the requested field or None, if accessing a non-existant field or a field that has not yet been initialized in the IndexedPieces.

Return type object or None

Raises `TypeError` if `field` is not a basestring.

`indexed_piece` Module

This model represents an indexed and analyzed piece of music.

class `vis.models.indexed_piece.`**`IndexedPiece`**(*pathname*, *opus_id=None*)

Bases: `object`

Hold indexed data from a musical score.

`get_data`(*analyzer_cls*, *settings=None*, *data=None*, *known_opus=False*)

Get the results of an Experimenter or Indexer run on this *`IndexedPiece`*.

Parameters

- **`analyzer_cls`** (*list of type*) – The analyzers to run, in the order they should be run.
- **`settings`** (*dict*) – Settings to be used with the analyzers.
- **`data`** (list of `pandas.Series` or `pandas.DataFrame`) – Input data for the first analyzer to run. If the first indexer uses a `Score`, you should leave this as `None`.
- **`known_opus`** (*boolean*) – Whether the caller knows this file will be imported as a `music21.stream.Opus` object. Refer to the “Note about Opus Objects” below.

Returns Results of the analyzer.

Return type `pandas.DataFrame` or list of `pandas.Series`

Raises `TypeError` if the `analyzer_cls` is invalid or cannot be found.

Raises `RuntimeError` if the first analyzer class in `analyzer_cls` does not use `Score` objects, and `data` is `None`.

Raises `OpusWarning` if the file imports as a `music21.stream.Opus` object and `known_opus` is `False`.

Raises `OpusWarning` if `known_opus` is `True` but the file does not import as an `Opus`.

Note about Opus Objects

Correctly importing `Opus` objects is a little awkward because we only know a file imports to an `Opus` *after* we import it, but an `Opus` should be treated as multiple `IndexedPiece` objects.

We recommend you handle `Opus` objects like this:

1. Try to call `get_data()` on the `IndexedPiece`.
2. If `get_data()` raises an `OpusWarning`, the file contains an `Opus`.
3. Call `get_data()` again with the `known_opus` parameter set to `True`.
4. `get_data()` will return multiple `IndexedPiece` objects, each corresponding to a `Score` held in the `Opus`.
5. Then call `get_data()` on the new `IndexedPiece` objects to get the results initially desired.

Refer to the source code for `vis.workflow.WorkflowManager.load()` for an example implementation.

`metadata`(*field*, *value=None*)
Get or set metadata about the piece.

Note: Some metadata fields may not be available for all pieces. The available metadata fields depend on the specific file imported. Unavailable fields return `None`. We guarantee real values for `pathname`, `title`, and `parts`.

Parameters

- **`field`** (*basestring*) – The name of the field to be accessed or modified.
- **`value`** (object or `None`) – If not `None`, the value to be assigned to `field`.

Returns The value of the requested field or `None`, if assigning, or if accessing a non-existant field or a field that has not yet been initialized.

Return type object or `None` (usually a basestring)

Raises `TypeError` if `field` is not a `basestring`.

Raises `AttributeError` if accessing an invalid `field` (see valid fields below).

Metadata Field Descriptions

All fields are taken directly from music21 unless otherwise noted.

Metadata Field	Description
alternativeTitle	A possible alternate title for the piece; e.g. Bruckner's Symphony No. 8 in C minor is known as "The German Michael."
anacrusis	The length of the pick-up measure, if there is one. This is not determined by music21.
composer	The author of the piece.
composers	If the piece has multiple authors.
date	The date that the piece was composed or published.
localeOfComposition	Where the piece was composed.
movementName	If the piece is part of a larger work, the name of this subsection.
movementNumber	If the piece is part of a larger work, the number of this subsection.
number	Taken from music21.
opusNumber	Number assigned by the composer to the piece or a group containing it, to help with identification or cataloguing.
parts	A list of the parts in a multi-voice work. This is determined partially by music21.
pathname	The filesystem path to the music file encoding the piece. This is not determined by music21.
title	The title of the piece. This is determined partially by music21.

Examples

```
>>> piece = IndexedPiece('a_sibelius_symphony.mei')
>>> piece.metadata('composer')
'Jean Sibelius'
>>> piece.metadata('date', 1919)
>>> piece.metadata('date')
1919
>>> piece.metadata('parts')
['Flute 1'{'Flute 2'{'Oboe 1'{'Oboe 2'{'Clarinet 1'{'Clarinet 2', ... ]
```

exception `vis.models.indexed_piece.`**`OpusWarning`**

Bases: `exceptions.RuntimeWarning`

The *`OpusWarning`* is raised by *`IndexedPiece.get_data()`* when `known_opus` is `False` but the file imports as a `music21.stream.Opus` object, and when `known_opus` is `True` but the file does not import as a `music21.stream.Opus` object.

Internally, the warning is actually raised by `IndexedPiece._import_score()`.

5.1.3 `workflow` Module

The `workflow` module holds the *`WorkflowManager`*, which automates several common music analysis patterns for counterpoint. The `TemplateWorkflow` class is a template for writing new `WorkflowManager` classes.

class `vis.workflow.`**`WorkflowManager`**(*pathnames*)

Bases: `object`

Parameters **`pathnames`** (list or tuple of basestring or *`IndexedPiece`*) – A list of pathnames.

The *`WorkflowManager`* automates several common music analysis patterns for counterpoint. Use the `WorkflowManager` with these four tasks:

- *`load()`*, to import pieces from symbolic data formats.

- `run()`, to perform a pre-defined analysis.
- `output()`, to output a visualization of the analysis results.
- `export()`, to output a text-based version of the analysis results.

Before you analyze, you may wish to use these methods:

- `metadata()`, to get or set the metadata of a specific `IndexedPiece` managed by this `WorkflowManager`.
- `settings()`, to get or set a setting related to analysis (for example, whether to display the quality of intervals).

You may also treat a `WorkflowManager` as a container:

```
>>> wm = WorkflowManager(['piece1.mxl', 'piece2.krn'])
>>> len(wm)
2
>>> ip = wm[1]
>>> type(ip)
<class 'vis.models.indexed_piece.IndexedPiece'>
```

load(*instruction='pieces'*, *pathname=None*)

Import analysis data from long-term storage on a filesystem. This should primarily be used for the `'pieces'` instruction, to control when the initial music21 import happens.

Use `load()` with an instruction other than `'pieces'` to load results from a previous analysis run by `run()`.

Note: If one of the files imports as a `music21.stream.Opus`, the number of pieces and their order *will* change.

Parameters

- **instruction** (*basestring*) – The type of data to load. Defaults to `'pieces'`.
- **pathname** (*basestring*) – The pathname of the data to import; not required for the `'pieces'` instruction.

Raises `RuntimeError` if the `instruction` is not recognized.

Instructions

Note: only `'pieces'` is implemented at this time.

- `'pieces'`, to import all pieces, collect metadata, and run `NoteRestIndexer`
- `'hdf5'` to load data from a previous `export()`.
- `'stata'` to load data from a previous `export()`.
- `'pickle'` to load data from a previous `export()`.

metadata(*index*, *field*, *value=None*)

Get or set a metadata field. The valid field names are determined by `IndexedPiece` (refer to the documentation for `metadata()`).

A metadatum is a salient musical characteristic of a particular piece, and does not change across analyses.

Parameters

- **`index`** (`int`) – The index of the piece to access. The range of valid indices is `0` through one fewer than the return value of calling `len()` on this *`WorkflowManager`*.
- **`field`** (`basestring`) – The name of the field to be accessed or modified.
- **`value`** (`object` or `None`) – If not `None`, the new value to be assigned to `field`.

Returns The value of the requested field or `None`, if assigning, or if accessing a non-existant field or a field that has not yet been initialized.

Return type `object` or `None`

Raises `TypeError` if `field` is not a `basestring`.

Raises `AttributeError` if accessing an invalid `field`.

Raises `IndexError` if `index` is invalid for this `WorkflowManager`.

`output`(*instruction*, *pathname=None*, *top_x=None*, *threshold=None*)

Output the results of the most recent call to *`run()`*, saved in a file. This method handles both visualizations and symbolic output formats.

Note: For LiliyPond output, you must have called *`run()`* with `count frequency` set to `False`.

Parameters

- **`instruction`** (*basestring*) – The type of visualization to output.
- **`pathname`** (*basestring*) – The pathname for the output. The default is `'test_output/output_result`. Do not include a file-type "extension," since we add this automatically. For the LilyPond experiment, if there are multiple pieces in the *`WorkflowManager`*, we append the piece's index to the pathname.
- **`top_x`** (*integer*) – This is the "X" in "only show the top X results." The default is `None`. Does not apply to the LilyPond experiment.
- **`threshold`** (*integer*) – If a result is strictly less than this number, it will be left out. The default is `None`. This is ignored for the `'LilyPond'` instruction. Does not apply to the LilyPond experiment.

Returns The pathname(s) of the outputted visualization(s). Requesting a histogram always returns a single string; requesting a score (or some scores) always returns a list.

Return type basestring or list of basestring

Raises `RuntimeError` for unrecognized instructions.

Raises `RuntimeError` if *`run()`* has never been called.

Raises `RuntiemError` if a call to R encounters a problem.

Raises `RuntimeError` with LilyPond output, if we think you called *`run()`* with `count frequency` set to `True`.

Instructions:

- `'histogram'`: a histogram. Currently equivalent to the `'R histogram'` instruction.
- `'LilyPond'`: each score with annotations for analyzed objects.
- **`'R histogram'`: a histogram with ggplot2 in R. Currently equivalent to the** `'histogram'` instruction. In the future, this will be used to distinguish histograms produced with R from those produced with other libraries, like matplotlib or bokeh.

- `'CSV'`: output a Series or DataFrame to a CSV file.
- `'Stata'`: output a Stata file for importing to R.
- `'Excel'`: output an Excel file for Peter Schubert.
- `'HTML'`: output an HTML table, as used by the VIS Counterpoint Web App.

Note: We try to prevent you from requesting LilyPond output if you called *`run()`* with `count frequency` set to `True` by raising a `RuntimeError` if `count frequency` is `True`, or the number of pieces is not the same as the number of results. It is still possible to call *`run()`* with `count frequency` set to `True` in a way we will not detect. However, this always causes *`output()`* to fail. The error will probably be a `TypeError` that says `object of type 'numpy.float64' has no len()`.

`run` (*instruction*)

Run an experiment's workflow. Remember to call *`load()`* before this method.

Parameters **`instruction`** (*basestring*) – The experiment to run (refer to "List of Experiments" below).

Returns The result of the experiment.

Return type `pandas.Series` or `pandas.DataFrame` or a list of lists of `pandas.Series`. If `'count frequency'` is set to False, the return type will be a list of lists of series wherein the containing list has each piece in the experiment as its elements (even if there is only one piece in the experiment, this will be a list of length one). The contained lists contain the results of the experiment for each piece where each element in the list corresponds to a unique voice combination in an unlabelled and unpredictable fashion. Finally each series corresponds the experiment results for a given voice combination in a given piece.

Raises `RuntimeError` if the `instruction` is not valid for this *`WorkflowManager`*.

Raises `RuntimeError` if you have not called *`load()`*.

Raises `ValueError` if the voice-pair selection is invalid or unset.

List of Experiments

- `'intervals'`: find the frequency of vertical intervals in 2-part combinations. All settings will affect analysis *except* `'n'`. No settings are required; if you do not set `'voice combinations'`, all two-part combinations are included.
- `'interval n-grams'`: find the frequency of n-grams of vertical intervals connected by the horizontal interval of the lowest voice. All settings will affect analysis. You must set the `'voice combinations'` setting. The default value for `'n'` is `2`.

`settings` (*index*, *field*, *value=None*)

Get or set a value related to analysis. The valid values are listed below.

A setting is related to this particular analysis, and is not a salient musical feature of the work itself.

Refer to *`run()`* for a list of settings required or used by each experiment.

Parameters

- **`index`** (int or `None`) – The index of the piece to access. The range of valid indices is `0` through one fewer than the return value of calling `len()` on this WorkflowManager. If `value` is not `None` and `index` is `None`, you can set a field for all pieces.
- **`field`** (*basestring*) – The name of the field to be accessed or modified.

- **value** (object or None) – If not None, the new value to be assigned to field.

Returns The value of the requested field or None, if assigning, or if accessing a non-existant field or a field that has not yet been initialized.

Return type object or None

Raises AttributeError if accessing an invalid field (see valid fields below).

Raises IndexError if index is invalid for this WorkflowManager.

Raises ValueError if index and value are both None.

Piece-Specific Settings

Pieces do not share these settings.

- **offset interval: If you want to run the *FilterByOffsetIndexer*, specify a value for this** setting. To avoid running the FilterByOffsetIndexer, set this to 0. setting that will become the quarterLength duration between observed offsets.
- filter repeats: If you want to run the *FilterByRepeatIndexer*, set this setting to True.
- voice combinations: If you want to consider certain specific voice combinations, set this setting to a list of a list of iterables. The following value would analyze the highest three voices with each other: '[[0,1,2]]' while this would analyze the every part with the lowest for a four-part piece: '[[0, 3], [1, 3], [2, 3]]'. This should always be a basestring that nominally represents a list (except the special values for 'all' parts at once or 'all pairs').

Shared Settings

All pieces share these settings. The value of index is ignored for shared settings, so it can be anything.

- n: As specified in *vis.analyzers.indexers.ngram.NGramIndexer.possible_settings*.
- continuer: Determines the way unisons that arise from sustained notes in the lowest voice are represented. Note that if the FilterByOffsetIndexer is used, the continuer won't get used. The default is 'dynamic quality' which sets to 'P1' if interval quality is set to True, and '1' if it is set to False. This is given directly to the NGramIndexer. Refer to *possible_settings*.
- interval quality: If you want to display interval quality, set this setting to True.
- simple intervals: If you want to display all intervals as their single-octave equivalents, set this setting to True.
- include rests: If you want to include 'Rest' tokens as vertical intervals, change this setting to True. The default is False.
- count frequency: When set to True (the default), experiments will return the number of occurrences of each token (i.e., "each interval" or "each interval n-gram"). When set to False, the moment-by-moment analysis of each piece is retained. We recommend you only request spreadsheet-formatted output when count frequency is False.

vis.workflow.**split_part_combo**(*key*)

Split a comma-separated list of two integer part names into a tuple of the integers.

Parameters **key** (*basestring*) – String with the part names.

Returns The indices of parts referred to by the key.

Return type tuple of int

```
>>> split_part_combo('5,6')
(5, 6)
>>> split_part_combo('234522,98100')
```

```
(234522, 98100)
>>> var = split_part_combo('1,2')
>>> split_part_combo(str(var[0]) + ',' + str(var[1]))
(1, 2)
```

CHAPTER 6

Indices and Tables

- genindex
- modindex
- search