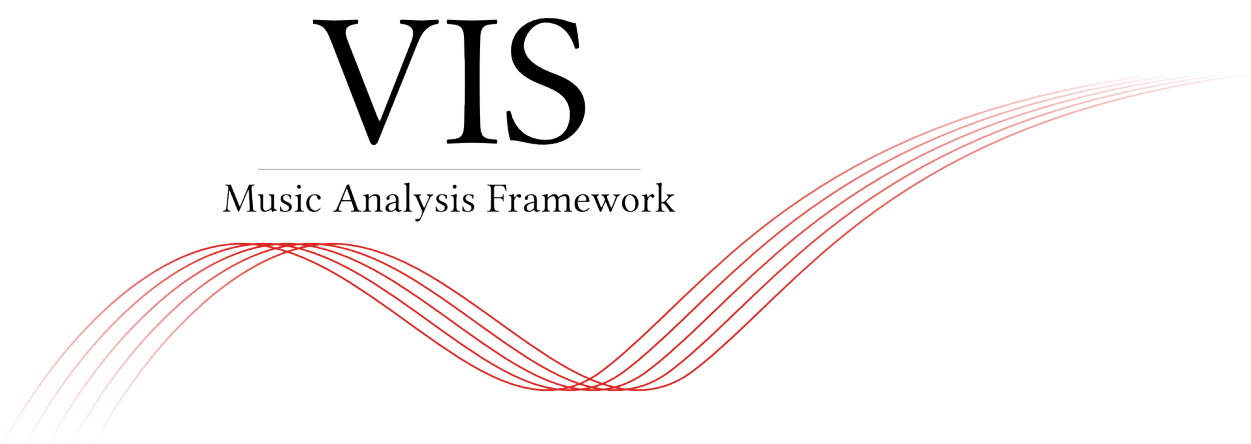
VIS Framework Documentation

Release 3.1.0

Christopher Antila

Sep 25, 2017

Contents



This is the API documentation for the VIS Framework for Music Analysis. The VIS Framework is a Python package that uses music21 and pandas to build a flexible and easy system for writing symbolic music analysis computer programs.

By providing a free-and-open solution (AGPLv3+) we hope to lower the barrier to empirical music analysis. The VIS Framework was developed at McGill University as part of the ELVIS Project (http://elvisproject.ca). VIS has been presented at conferences, including the International Society for Music Information Retrieval and the Society for Music Theory.

The API is written for programmers interested in symbolic music information retrieval (i.e., music theory analysis performed by computers). Musicologists and programmers alike may access our VIS client at vis-rodan.simssa.ca, and some of our research findings from the ELVIS Project's website.

The rest of the documentation discusses the VIS Framework's architecture, how to install and use the framework, and how to add your own analysis tasks.

CHAPTER 1

One-Paragraph Introduction

The VIS Framework uses two data models (*IndexedPiece* and *AggregatedPieces*) to fetch results for one or multiple pieces, respectively. Call their *get_data()* method with a list of analyzer classes to run, and a dictionary with their settings. After you develop an analysis workflow, add it to the *WorkflowManager* for more consistent operation.

CHAPTER 2

Table of Contents

Learn about and Install the VIS Framework

Design Principles

Three Simple Components

In essence, the VIS Framework is built on three simple components: *analyzers* make music analysis decisions; *models* run analyzers on a score; the *WorkflowManager* determines the order analyzers are run. In other words, the three components are about analysis decisions, making the decisions happen, and ordering the decision-happening.

Consider this example. A common task for VIS is to count the number of vertical intervals in a piece of music. You ask the *WorkflowManager* to run this query. The *WorkflowManager* knows the steps involved: (1) name the notes in a score, (2) find the vertical intervals between simultaneous notes, and (3) count the number of occurrences of every interval. For each step, the *WorkflowManager* asks a *model* for results. The model represents a piece, and it knows, for example, how to find simultaneous events, and how to meaningfully organize the results of an analyzer. Finally, an *analyzer* makes a single type of music analysis decision, for a single moment. For example, the analyzer called *IntervalIndexer* takes two note names and determines the interval between them.

For a relatively simple music analysis task like counting the number of vertical intervals, these three components may seem anything *but* simple. For more complicated music analysis tasks, the Framework's architecture begins to pay off. Whether finding contrapuntal modules, analyzing harmonic function, or anything else, these components will be enough to get the job done. To design a new query (say, if you want to label chordal dissonances) you only need to add one analyzer for every analysis decision, then tell the WorkflowManager the order the analyzers should run. Complicated analysis tasks will always be complicated, but VIS provides a solid, predictable Framework for any task, allowing you to focus on what's special about your query, rather than on making sure you remember how to load pieces properly.

Three Levels of Interaction

Because of its flexibility, you may choose to interact with the VIS Framework on one of three levels, depending on the flexibility required for your task.

If you simply want to use VIS for one of its built-in queries, like finding vertical intervals or contrapuntal modules, you can use VIS **as a program**. You may do this through a graphical interface like the Counterpoint Web App or through the Python shell directly, as described in use_as_a_program.

If the built-in `WorkflowManager` does not provide the workflow you need, but you can still accomplish your query with the built-in analyzers, you can use VIS **as a library**. For example, you may wish to analyze melodic patterns with *n*-grams, as described in *Tutorial: Use N-Grams to Find Melodic Patterns*.

Finally, if your query cannot be implemented using the built-in analyzers, you can use VIS **as a framework**, adding analyzer modules and modifying the `WorkflowManager` as necessary. For example, you may wish to DO WHATEVER WILL BE DESCRIBED IN THIS CROSS-REF.

A More Detailed Look

Two Types of Analyzers

Analyzers make music analysis decisions. The VIS Framework has two types of analyzers: indexers and experimenters.

Indexers use a `music21.stream.Score`, or a `pandas.DataFrame` from another indexer, to perform a music analytic calculation. The output of any indexer can be sensibly attached to a specific moment of a piece. That is, indexers are for events that "happen" at an identifiable time. Indexers may be relatively simple, like the *`IntervalIndexer`*, which accepts an index of the notes and rests in a piece, transforming it into an index of the vertical intervals between all the pairs of parts. Indexers may also 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 scale degrees, the harmonic function of a chord, the figured bass signature of a simultaneity, or the moment of a modulation.

Experimenters always start with a `DataFrame` produced by another analyzer, producing results that cannot be sensibly attached to a specific moment of a piece. That is, experimenters are for characteristics of a piece (or movement) as a whole. Experiments may be relatively simple, like the *`FrequencyExperimenter`*, which counts the number of occurrences of the objects in a `DataFrame`. Experimenters may also be complicated, like one that produces a Markov transition model of figured bass signatures.

The distinction between indexers and experimenters helps to organize valid workflows. Analyses may flow from indexer to indexer, from indexer to experimenter, and from experimenter to experimenter. However, an analysis may not move from an experimenter to an indexer; once moment-specific information is lost, it cannot be recovered. (The exception to the rule: indexers may use data from experimenters—as long as they also use data from another indexer or a `Score`).

When designing your own analyzers, we encourage you to avoid the temptation to include many analysis steps in the same analyzer, and instead to follow the design pattern set out with our own analyzers and our `TemplateIndexer` and `TemplateExperimenter`. Following this design pattern helps ensure your program is easy to test, and therefore more trustworthy. In addition, you may be able to contribute valuable new analyzer modules that will help other scholars get started with VIS more easily.

If required, you may use an analyzer to run external programs, possibly written in a different programming language. For example, the `RBarChart` experimenter runs a program in the R language, using the *ggplot2* library to produce a bar chart. Another example is the `LilyPondExperimenter`, which uses the external `outputlilypond` Python module to produce a file for LilyPond, a C program, which that module calls directly.

Two Types of Models

VIS uses two types of models: *`IndexedPiece`* and *`AggregatedPieces`*. These models represent a single piece (or movement), and a group of pieces (and movements), respectively. In a typical application, you will write analyzers but never call their methods directly. On the other hand, you will almost never modify the models, but call their

methods very often. Models know how to run analyzers on the piece or pieces they represent, how to import music21 `Score` objects safely and efficiently, and how to find and access metadata. The models also perform some level of automated error-handling and data-coordination. In the future, the models may also help coordinate multiprocessing or results-caching, and they should be able to do this without a change in the API.

Known Issues and Limitations

- 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`*.
- Limitation: 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 the solutions have not been fully implemented yet.

Install the Framework

You must install the VIS Framework before you use it. Our recommended installation method depends on how you plan to use VIS.

If you plan to use VIS as a program or a library, you should install the Framework for *deployment*. This includes users who will conduct their own analyses with the built-in modules, and users who will host an instance of the Counterpoint Web App. A deployment installation will always be "stable" software, meaning that the software has been tested more thoroughly, and we can offer a higher assurance of correctness.

If you plan to use VIS as a framework, you should install the Framework for *development*. This includes users who will write their own analyzers for the VIS Framework, and users who will modify existing modules. A development installation allows you to use the most up-to-date versions of VIS, and even to suggest your improvements for inclusion in a VIS release, although these versions may crash more often and be less correct than deployment versions.

You may safely install the VIS Framework multiple times on the same computer, including a mixture of deployment and development installations.

Python 3

With the release of VIS Framework 2.1, VIS can run and is tested with Python 2.7, 3.3, and 3.4. We recommend a Python 3 release for reasons of future compatibility. The music21 developers have announced plans to drop support for Python 2.7 in calendar year 2016, at which point VIS will also drop support for Python 2.7. Python 3.4 offers several major enhancements over Python 3.3, including improved speed of execution, so we recommend you use Python 3.4 if possible.

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
```

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 (listed in the "requirements.py" file), 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. 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 administrator privileges, though we recommend using virtualenv. 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 the 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.

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.

Optional: Test the Framework Before Use

You may wish to run the VIS Framework's automated test suite before you use it, to confirm the installation was successful and is working as expected.

To run the test suite:

1. Change into the main VIS directory (this will depend on your installation, but it will be similar to `/usr/lib/python3.4/site-packages/vis-framework` or `/home/crantila/virtualenvs/vis-virtenv3/lib/python3.4/site-packages/vis-framework`.
2. If you installed VIS with a virtualenv, activate the virtualenv (e.g., `source ~/virtualenvs/vis-virtenv3/bin/activate`).
3. Run the test script: `python run_tests.py`.
4. Python prints a `.` character for every successful test, and an error or warning for every test that fails.

If Tests Fail

If you are using a deployment release and one or more of the automated tests fail, the most likely reason is that the installation was somehow unsuccessful. This may result from using untested versions of a dependency, unexpectedly troublesome environment variables, or other factors. If you used distribution packages (i.e., did not use virtualenv) and tests fail, we do recommend in that case installing with virtualenv. Both with and without virtualenv, if tests fail, ensure you are using the exact package versions indicated in `requirements.txt`, since they are the versions used by the development team.

If you are using a development release and automated tests fail, check if those tests also fail on our "Travis CI" test server. If so, the VIS development team is already aware of the failure and working toward a fix, since Travis will notify us by email if a test fails. You may also try the version-related advice in the previous paragraph, though you may also wish to prepare a fix for the problem and submit it to us as a GitHub pull request.

Optional: 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 test with 3.0.x versions of R.

If you use a "Windows" computer, download a pre-compiled R interpreter from http://cran.r-project.org. If you use an "OS X" computer, you may download a pre-compiled binary or use a package from Homebrew, Fink, or MacPorts. If you use a "Linux" computer (or "BSD," etc.), check your package manager for an appropriate version of R. You may have difficulty searching for "R," as in `yum search 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. For all operating systems, if you encounter a problem, the R manuals offer extensive help, but require careful attention.

After you install R, you must install the *ggplot2* package. If you installed R with your package manager, we recommend you search for and use the "ggplot" package, if one is provided. (Distribution packages offer maximum compatibility, and take advantage of centralized software updates).

Use the following instructions, which work on all operating systems, if you do not have a distribution package for *ggplot2*.

1. Start R (with superuser privileges, if not using Windows).
2. Run the following command to install *ggplot2*:

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

3. Run the following program to test R and *ggplot2*:

```
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()
```

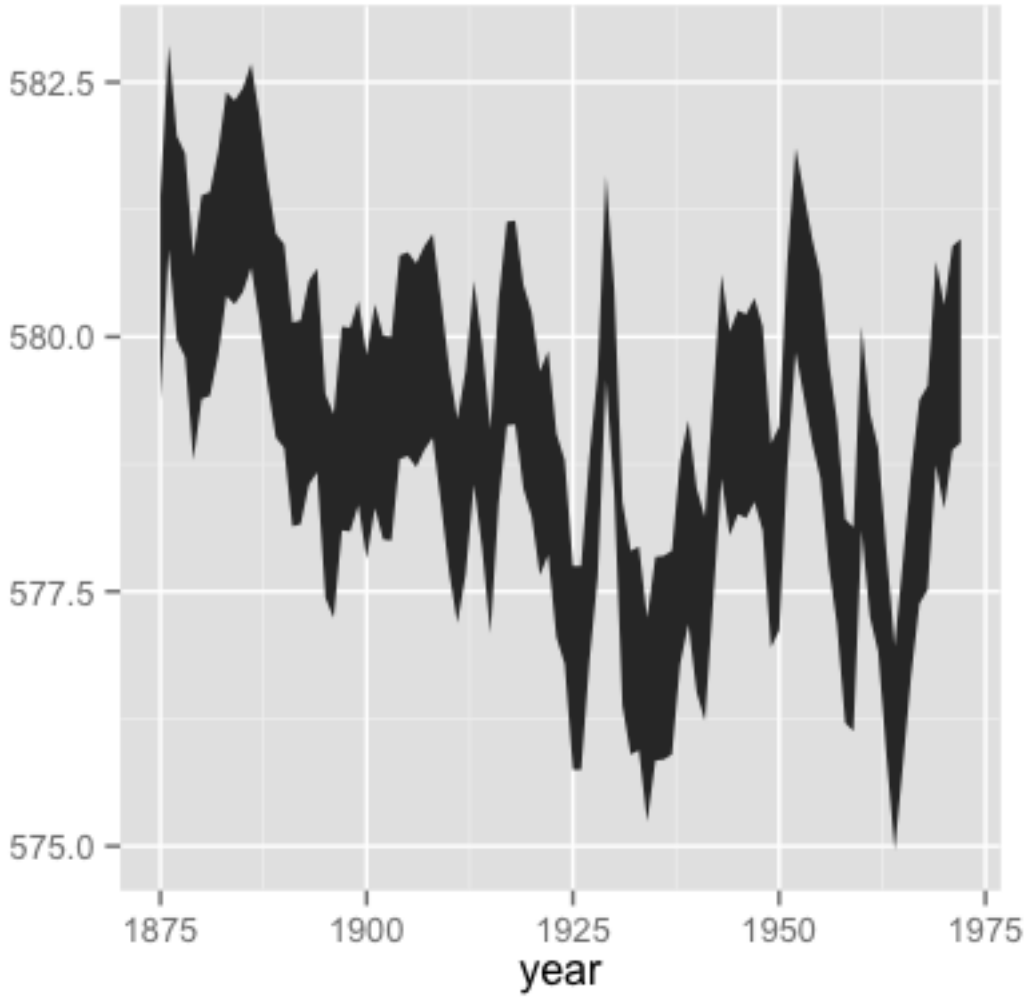


Fig. 2.1: Image credit: taken from the "ggplot2" documentation on 26 November 2013; reused here under the GNU General Public License, version 2.

Optional: Install LilyPond for Annotated Scores

If you wish to produce annotated scores with the VIS Framework, you must install LilyPond.

The *outputlilypond* module, used by VIS, is targeted for 2.18.x versions of LilyPond, though it should also work with 2.16.x versions. We do note recommend versions numbered 2.15.x, 2.17.x, and 2.19.x, since these are intended only for LilyPond developers, and they may crash or produce incorrect output. If possible, we recommend you install LilyPond with your distribution's package management system, or (on "OS X") with a package manager such as Homebrew. "Windows" users, and users who do not have a 2.16.x or 2.18.x version of LilyPond available from their package manager, may download and install a pre-compiled version of LilyPond from their website, lilypond.org.

We very strongly discourage users from compiling LilyPond themselves. There is very little chance that the benefits of self-compilation will outweigh the mental distress.

Programming Tutorials for VIS

Tutorial: Use *N*-Grams to Find Melodic Patterns

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

Develop a Question

Our research question involves numerically comparing melodic styles of multiple composers. To help focus our findings on the differences between *composers*, our test sets should consist of pieces that are otherwise as similar as possible. One of the best ways to compare styles is using patterns, which are represented in the VIS Framework as *n*-grams: a unit of *n* objects in a row. While the Framework's *n*-gram functionality is fairly complex, in this tutorial we will focus on simple *n*-grams of melodic intervals, which will help us find melodic patterns. The most frequently occurring melodic patterns will tell us something about the melodic styles of the composers under consideration: we will be pointed to some similarities and some differences that, taken together, will help us refine future queries.

Since *n*-grams are at the centre of the preliminary investigation described in this tutorial, we will use the corresponding `NGramIndexer` to guide our development. We must answer two questions:

1. What data will the `NGramIndexer` require to find melodic patterns?
2. What steps are required after the `NGramIndexer` to produce meaningful results?

We investigate these two questions in the following sections.

What Does the NGramIndexer Require?

To begin, try reading the documentation for the `NGramIndexer`. At present, this Indexer is the most powerful and most complicated module in the VIS Framework, and as such it may pose difficulties and behave in unexpected ways. For this tutorial we focus on the basic functionality: the "n" and "vertical" settings.

TODO: continue revising here

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 `['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:

```
1  from vis.analyzers.indexers import noterest, interval, ngram
2  from vis.models.indexed_piece import IndexedPiece
3
4  # prepare inputs and output-collectors
5  pathnames = [list_of_pathnames_here]
6  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], interval_settings)
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.

How Shall We Prepare 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:

```
1  from vis.analyzers.indexers import noterest, interval, ngram
2  from vis.analyzers.experimenters import frequency, aggregator
3  from vis.models.indexed_piece import IndexedPiece
4  from vis.models.aggregated_pieces import AggregatedPieces
5  from pandas import DataFrame
6
7  # prepare inputs and output-collectors
8  pathnames = [list_of_pathnames_here]
9  ind_ps = [IndexedPiece(x) for x in pathnames]
```

```
10 interval_settings = {'quality': True}
11 ngram_settings = {'vertical': [0], 'n': 3}  # change 'n' as required
12 ngram_freqs = []
13
14 # prepare for and run the NGramIndexer
15 for piece in ind_ps:
16     intervals = piece.get_data([noterest.NoteRestIndexer, interval.
   ↪HorizontalIntervalIndexer], interval_settings)
17     for part in intervals:
18         ngram_freqs.append(piece.get_data([ngram.NGramIndexer, frequency.
   ↪FrequencyExperimenter], ngram_settings, [part]))
19
20 # aggregate results of all pieces
21 agg_p = AggregatedPieces(ind_ps)
22 result = agg_p.get_data([aggregator.ColumnAggregator], [], {}, ngram_freqs)
23 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)`.

Next Steps

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.

Tutorial: Use the WorkflowManager

The script developed in *Tutorial: Use N-Grams to Find Melodic Patterns* is suitable for users doing exploratory work in an interactive Python shell. When a query becomes regularized and you want it to be easily repeatable, or if you are an application developer making a graphical interface (whether on the Web or in a desktop application) you can take advantage of a further layer of abstraction offered by our *`WorkflowManager`*. The `WorkflowManager` is designed as the point of interaction for end-user applications, providing a consistent interface and reference implementations of the steps involved in all queries. While every new query will involve modifying a portion of the `run()` method's code, you may be able to re-use the existing input and output methods without change.

The `WorkflowManager`'s documentation describes its functionality:

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

Warning: The WorkflowManager is deprecated as of VIS 3.0 and will be entirely removed in VIS 4.0. Most

of its functionality still works with VIS 3.0 but this is not guaranteed and it is no longer being supported in development.

Parameters **pathnames** (list or tuple of string 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 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'>
```

Port a Query into the WorkflowManager

Porting an existing query to the WorkflowManager involves fitting its code into the appropriate pre-existing methods. The load() method prepares IndexedPiece objects and metadata by loading files for analysis. The output() method outputs query results to a variety of formats, including spreadsheets, charts, and scores. You will not usually need to modify load(), and you may not need to modify output() either.

The majority of development work will be spent in the run() method or its related hidden methods (like the _intervs(), _inteval_ngrams(), and other methods that are included in the default WorkflowManager).

TODO: continue revising here.

When I add my new query's logic to the run() method, I get this:

```
def run(self):
    ngram_settings = {'vertical': [0], 'n': self.settigns(None, 'n')}
    ngram_freqs = []

    for i, piece in enumerate(self._data):
        interval_settings = {'quality': self.settings(i, 'interval quality')}
        intervals = piece.get_data( \
            [noterest.NoteRestIndexer, interval.HorizontalIntervalIndexer], \
            interval_settings)
        for part in intervals:
            ngram_freqs.append( \
                piece.get_data([ngram.NGramIndexer, frequency.FrequencyExperimenter],␣
↪\
                               ngram_settings, \
                               [part]))

```

```
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: Use N-Grams to Find Melodic Patterns*, use the `WorkflowManager` like this:

```
1  from vis.workflow import WorkflowManager
2
3  pathnames = [list_of_pathnames]
4  work = WorkflowManager(pathnames)
5  work.load('pieces')
6  for i in xrange(len(work)):
7      work.settings(i, 'quality', True)
8  work.run()
9  work.export('CSV', 'output_filename.csv')
```

This script actually does more than the program in *Tutorial: Use N-Grams to Find Melodic Patterns* because `export()` "converts" the results to a `DataFrame`, sorts, and outputs the results.

API Specification

vis

analyzers Package

`analyzers` Package

Indexers produce an "index" of a symbolic musical score. Each index provides one type of data, and each event 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 of the 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 = []

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 six.string_types*) – Indices of the parts or the part combinations, or another descriptive label as described in the indexer subclass documentation.
- **indices** (list of pandas.Series or a pandas.DataFrame) – 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.**series_indexer**(*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

- **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 some other string.

Returns The pipe_index argument and the new index. The new index is a pandas.Series where every element is a 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.

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`

Example:

import music21 from vis.analyzers.indexers import noterest from vis.analyzers.experimenters import aggregator, frequency

score = music21.converter.parse('example.xml') notes = noterest.NoteRestIndexer(score).run()

freqs = frequency.FrequencyExperimenter(notes).run() agg = aggregator.ColumnAggregator(freqs).run() print(agg)

`barchart` 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`

Example:

import music21 from vis.analyzers.indexers import noterest from vis.analyzers.experimenters import aggregator, frequency

score = music21.converter.parse('example.xml') notes = noterest.NoteRestIndexer(score).run()

freqs = frequency.FrequencyExperimenter(notes).run() agg = aggregator.ColumnAggregator(freqs).run() print(agg)

dendrogram 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`

Example:

import music21 from vis.analyzers.indexers import noterest from vis.analyzers.experimenters import aggregator, frequency

score = music21.converter.parse('example.xml') notes = noterest.NoteRestIndexer(score).run()

freqs = frequency.FrequencyExperimenter(notes).run() agg = aggregator.ColumnAggregator(freqs).run() print(agg)

`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`

Example:

import music21 from vis.analyzers.indexers import noterest from vis.analyzers.experimenters import frequency

score = music21.converter.parse('example.xml') notes = noterest.NoteRestIndexer(score).run()

freqs = frequency.FrequencyExperimenter(notes).run() print(freqs)

`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 strings 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

active_voices Module

class vis.analyzers.indexers.active_voices.**ActiveVoicesIndexer**(*score*, *settings=None*)

Bases: *vis.analyzers.indexer.Indexer*

Indexer that counts the number of voices active at each offset. It can either find all voices sounding, or only the voices that are attacking, depending on the settings passed.

Call this indexer via the get_data() method of either an indexed_piece object or an aggregated_pieces object (see example below). If nothing is passed in the 'data' argument of the call to get_data(), then the default is to process the NoteRestIndexer results of the indexed_piece in question. You can pass some other DataFrame in the 'data' argument, but it is discouraged.

Parameters

- **'attacked'** (*boolean*) – When true, only counts the voices that are attacking at each offset. Defaults to false.
- **'show_all'** (*boolean*) – When true, shows the results at all offsets, even if there is not change. Defaults to false.

Examples:

Prepare an indexed piece:

```
>>> from vis.models.indexed_piece import Importer
>>> ip = Importer('path_to_piece.xml')
```

Get the ActiveVoicesIndexer results with the default settings:

```
>>> ip.get_data('active_voices')
```

Get the ActiveVoicesIndexer results with specified settings:

```
>>> av_setts = {
        'attacked': True,
        'show_all': True
    }
>>> ip.get_data('active_voices', settings=av_setts)
```

default_settings = {'show_all': False, 'attacked': False}

possible_settings = ['attacked', 'show_all']

required_score_type = 'pandas.DataFrame'

run()

Returns new index of the active voices in the piece.

Return type pandas.DataFrame

vis.analyzers.indexers.active_voices.**indexer1**(*x*)
Used internally by the *ActiveVoicesIndexer* to count individualevents.

approach Module

class vis.analyzers.indexers.approach.**ApproachIndexer**(*score*, *settings=None*)
Bases: *vis.analyzers.indexer.Indexer*

Using OverBassIndexer and FermataIndexer results, finds cadences as lists of events in the approach to a fermata.

Call this indexer via the get_data() method of either an indexed_piece object or an aggregated_pieces object (see example below).

Parameters

- **'length'** (*int*) – The length of the cadence, or how many events happen before a fermata.
- **'voice'** (*str or int*) – The voice in which you want to look for fermatas. The default value for this is 'all'.

Example:

Prepare an indexed piece and import pandas:

```
>>> from vis.models.indexed_piece import Importer
>>> import pandas
>>> ip = Importer('path_to_piece.xml')
```

Prepare OverBassIndexer and FermataIndexer results. For more specific advice on how to do this, please see the documentation of those two indexers. These two DataFrames should be passed as a list. For simplicity, including the FermataIndexer results is optional, and this example shows how to use the ApproachIndexer without explicitly providing the FermataIndexer results, so the 'data' argument is a singleton list.

```
>>> overbass_input_dfs = [ip.get_data('noterest'),
        ip.get_data('vertical_interval')]
>>> ob_setts = {
        'type': 'notes'
    }
>>> overbass = ip.get_data('over_bass', data=overbass_input_dfs,
        settings=ob_setts)
```

Get the ApproachIndexer results with specified settings:

```
>>> approach_setts = {'length': 3}
>>> ip.get_data('approach', data=[overbass], settings=approach_setts)
```

possible_settings = ['length', 'voice']

required_score_type = 'pandas.DataFrame'

run()
Makes a new index of the approaches to fermatas in the piece.

Returns A `DataFrame` of the approaches.

Return type `pandas.DataFrame`

`contour` Module

`vis.analyzers.indexers.contour.`**`COM_matrix`**(*contour*)
Creates a matrix representing the contour given.

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

Indexes the contours of a given length in a piece, where contour is a way of numbering the relative heights of pitches, beginning at 0 for the lowest pitch.

Call this indexer via the `get_data()` method of either an `indexed_piece` object or an `aggregated_pieces` object (see example below). If nothing is passed in the 'data' argument of the call to `get_data()`, then the default is to process the `NoteRestIndexer` results of the `indexed_piece` in question. You can pass some other DataFrame to the 'data' argument, but it is discouraged.

Parameters **`'length'`** (*int*) – This is the length of the contour you want to look at.

Example:

Prepare an indexed piece:

```
>>> from vis.models.indexed_piece import Importer
>>> ip = Importer('path_to_piece.xml')
```

Get the `ContourIndexer` results with specified settings and processing the notes and rests:

```
>>> notes = ip.get_data('noterest')
>>> contour_setts = {'length': 3}
>>> ip.get_data('contour', data=notes, settings=contour_setts)
```

`possible_settings` = ['length']

`required_score_type` = 'pandas.DataFrame'

`run`()
Makes a new index of the contours in the piece.

Returns A `DataFrame` of the contours.

Return type `pandas.DataFrame`

`vis.analyzers.indexers.contour.`**`compare`**(*contour1*, *contour2*)
Additional method to compare `COM_matrices`.

`vis.analyzers.indexers.contour.`**`getContour`**(*notes*)
Method used internally by the `ContourIndexer` class to convert pitches into contour numbers.

`dissonance` Module

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

Indexer that locates vertical dissonances between pairs of voices in a piece. It then categorizes intervals as consonant or dissonant and in the case of fourths (perfect or augmented) and diminished fifths it examines the other parts sounding with that fourth or fifth (if there are any) to see if the interval can be considered consonant.

This dissonance analysis allows for the assignment of a dissonance type name or a consonance label for each voice at each offset. This last step is the DataFrame that gets returned. The score type must be a list of dataframes of the results of the following indexers (order matters): beatstrength, duration, horizontal, vertical. To simplify things, however, it is better to use the `get_data()` on an `indexed_piece` object to get results from the dissonance indexer as per the example below. The results of the disonance indexer are in the form of one cell in the resultant dataframe in each part at every offset where there is a new note, rest, or chord onset in any part. These cells contain one-character strings which are an abbreviation of what the dissonance type for that part at that moment is. The abbreviations correspoond to the following dissonance types:

- **'Q': Dissonant third quarter (special type of accented passing** tone)
- 'D': Descending passing tone
- 'R': Ascending ("rising") passing tone
- 'L': Lower neighbour
- 'U': Upper neighbour
- 'S': Suspension
- 'F': Fake suspension
- 'f': Diminished fake suspension
- 'A': Anticipation
- 'C': Nota cambiata
- 'H': Chanson idiom
- 'E': Echappée (escape tone)
- **'-': Either no dissonance, or the part in question is not** considered to be the dissonant note of the dissonance it's in

Example: >>> from vis.models.indexed_piece import Importer >>> ip = Importer('symbolic_notation_file_location.xml') >>> ip.get_data('dissonance')

check_4s_5s(*pair_name*, *iloc_indx*, *suspect_diss*, *simuls*)

This function evaluates whether P4's, A4's, and d5's should be considered consonant based whether or not the lower voice of the suspect_diss forms an interval that causes us to deem the fourth or fifth consonant, as determined by the cons_makers list below. The function should be called once for each potentially consonant fourth or fifth. :param pair_name: Name of pair that has the potentially

consonant fourth or fifth.

Parameters

- **iloc_indx** (*Integer.*) – Pandas iloc number of interval's row in dataframe.
- **suspect_diss** (*String.*) – Interval name with quality and direction (i.e. nothing or '-') that corresponds to the fourth or fifth to be examined.

Returns string representing analysis of fourth or fifth in question where a 'C' or 'D' has been prepended to the interval name with quality to show that it was considered consonant or dissonant respectively.

Return type string

classify(*indx*, *pair*, *event*, *prev_event*)

Checks the dissonance definitions to find a suitable label for the dissonance passed. If no identifiable dissonance type matches, returns an unknown dissonance label. Omits checking the pair if either voice was previously given a known dissonance label still in vigour at the given offset. It only takes its four arguments

to pass them on to the dissonance-type functions it calls. :returns: A 5-tuple that can be unpacked to assign separate

labels for each voice in the pair. If none of the known dissonance types were detected the 5-tuple will have the labels to mark one or both of the voices as dissonant.

Return type tuple

required_score_type = 'pandas.DataFrame'

run()

Make a new index of the piece which consists of a DataFrame with as many columns as there are voices in the piece. The index is the offset of the dissonance analyses. Another DataFrame (`diss_ints`) is calculated but not returned. It is in the same format as the `IntervalIndexer` (i.e. a DataFrame of Series where each series corresponds to the intervals in a given voice pair). The difference between this and the interval indexer is that this one figures out whether fourths or diminished fifths should be considered consonant for the purposes of dissonance classification. diss_ints is not calculated separately because it essentially consists of the same loop needed for dissonance classification. :returns: A `DataFrame` of the new indices. The columns

have a `MultiIndex`.

Return type `pandas.DataFrame`

`fermata` Module

Index fermatas.

class `vis.analyzers.indexers.fermata.`**FermataIndexer**(*score*, *settings=None*)

Bases: *vis.analyzers.indexer.Indexer*

Index `Fermata`.

Finds :class:'~music21.expressions.Fermata's.

Example:

```
>>> from vis.models.indexed_piece import Importer
>>> ip = Importer('pathnameToScore.xml')
>>> ip.get_data('fermata')
```

required_score_type = 'stream.Part'

`vis.analyzers.indexers.fermata.`**indexer_func**(*event*)

Used internally by *FermataIndexer*. Inspects

Note and **Rest** and returns `u'Fermata'` if a fermata is associated, else NaN.

Parameters **event** (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 contains a `Fermata`, string `u'Fermata'` is returned. Else, NaN.

Return type str

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. Alternatively you could provide the results of the :class:'~vis.analyzers.offset.FilterByOffsetIndexer' if you want to check for horizontal intervals at regular durational intervals. These settings apply to the *HorizontalIntervalIndexer* *in addition to* the settings available from the *IntervalIndexer*.

Parameters

- **'simpleor compound'** (*str*) – Whether intervals should be represented in their single-octave form (either 'simple' or 'compound').
- **or string 'quality'** (*bool*) – Whether to display diatonic intervals without quality (either False or "diatonic no quality"), diatonic intervals with quality (either True or 'diatonic with quality'), chromatic intervals (use 'chromatic'), or interval class intervals (use 'interval class').
- **'directed'** (*boolean*) – Whether we distinguish between which note is higher than the other. If True (default), prepends a '-' before everything else if the first note passed is higher than the second.
- **'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.
- **'mp'** (*boolean*) –

  Multiprocesses when True (default) or processes serially when False.

  Example:

```
>>> from vis.models.indexed_piece import Importer
>>> settings = {
        'quality': 'interval class',
        'simple or compound': 'simple',
        'directed': False
    }
>>> ip = Importer('pathnameToScore.xml')
>>> ip.get_data('horizontal_interval', settings)
```

default_settings = {'directed': True, 'simple or compound': 'compound', 'horiz_attach_later': False, 'quality': Fal

run()

Make a new index of the piece.

Returns The new indices. Refer to the example below.

Return type pandas.DataFrame

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`. The settings for the *IntervalIndexer* are as follows:

Parameters

- **'simpleor compound'** (*str*) – Whether intervals should be represented in their single-octave form (either `'simple'` or `'compound'`).
- **or string 'quality'** (*bool*) – Whether to display diatonic intervals without quality (either False or "diatonic no quality"), diatonic intervals with quality (either True or 'diatonic with quality'), chromatic intervals (use 'chromatic'), or interval class intervals (use 'interval class').
- **'directed'** (*boolean*) – Whether we distinguish between which note is higher than the other. If True (default), prepends a '-' before everything else if the first note passed is higher than the second.
- **'mp'** (*boolean*) – Multiprocesses when True (default) or processes serially when False.

Example:

```
>>> from vis.models.indexed_piece import Importer
>>> settings = {
        'quality': 'chromatic',
        'simple or compound': 'simple',
        'directed': True
    }
>>> ip = Importer('pathnameToScore.xml')
>>> ip.get_data('vertical_interval', settings)
```

default_settings = {'directed': True, 'simple or compound': 'compound', 'quality': False, 'mp': True}

required_score_type = 'pandas.DataFrame'

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`

class vis.analyzers.indexers.interval.**IntervalReindexer**(*score*, *settings=None*)
Bases: *vis.analyzers.indexers.interval.HorizontalIntervalIndexer*

This indexer is only meant to ever be called indirectly to simplify caching of the results of the IntervalIndexer and the *HorizontalIntervalIndexer*. It takes the most information-rich type of interval analysis we offer (i.e. compound, directed intervals with diatonic quality) and re-indexes them to match whatever settings the user has requested. This is much faster, because it takes an entire interval as its input, rather than two notes, and therefore there are considerably fewer memos in its memoization scheme.

run()

meter Module

Indexers for metric concerns.

class vis.analyzers.indexers.meter.**DurationIndexer**(*score*, *part_streams*)
Bases: *vis.analyzers.indexer.Indexer*

Make an index of the durations of all `Note`, `Rest`, and `Chord` objects. These are calculated based on the difference in index positions of consecutive events.

Note: Unlike nearly all other indexers, this indexer returns a `Series` of `float` objects rather than `unicode` objects. Also unlike most other indexers, this indexer does not have an indexer func.

Example:

```
>>> from vis.models.indexed_piece import Importer
>>> ip = Importer('pathnameToScore.xml')
>>> ip.get_data('duration')
```

required_score_type = 'pandas.DataFrame'

run()
Make a new index of the piece.

Returns The new indices of the durations of each note or rest event in a score. Note that each item is a float, rather than the usual basestring.

Return type `pandas.DataFrame`

class vis.analyzers.indexers.meter.**MeasureIndexer**(*score*)
Bases: *vis.analyzers.indexer.Indexer*

Make an index of the measures in a piece. Time signatures changes do not cause a problem. Note that unlike most other indexers this one returns integer values >= 0. Using music21's part.measureTemplate() function is an alternative but it turned out to be much less efficient to looping over the piece and doing it this way makes this indexer just like all the other stream indexers in VIS.

Example:

```
>>> from vis.models.indexed_piece import Importer
>>> ip = Importer('pathnameToScore.xml')
>>> ip.get_data('measure')
```

required_score_type = 'pandas.DataFrame'

class vis.analyzers.indexers.meter.**NoteBeatStrengthIndexer**(*score*)
Bases: *vis.analyzers.indexer.Indexer*

Make an index of the `beatStrength` for all `Note`, `Rest`, and `Chord` objects.

Note: Unlike nearly all other indexers, this indexer returns a `Series` of `float` objects rather than `unicode` objects.

Example:

```
>>> from vis.models.indexed_piece import Importer
>>> ip = Importer('pathnameToScore.xml')
>>> ip.get_data('beat_strength')
```

required_score_type = 'pandas.DataFrame'

vis.analyzers.indexers.meter.**beatstrength_ind_func**(*event*)
Used internally by *NoteBeatStrengthIndexer*. Convert `Note` and `Rest` objects into a string.

Parameters **event** – A music21 note, rest, or chord object which get queried for its beat strength.
:type event: A music21 note, rest, or chord object or NaN.

Returns The `beatStrength` of the event which is dependent on the prevailing time signature.

Return type float

`vis.analyzers.indexers.meter.`**measure_ind_func**(*event*)
The function that indexes the measure numbers of each part in a piece. Unlike most other indexers, this one returns int values. Measure numbering starts from 1 unless there is a pick-up measure which gets the number 0. This can handle changes in time signature without problems.

Parameters **event** (*A music21 measure object or NaN.*) – A music21 measure object which get queried for its "number" attribute.

Returns The number attribute of the passed music21 measure.

Return type int

ngram Module

Indexer to find k-part any-object n-grams. This file is a re-implimentation of the previous ngram_indexer.py file.

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. The "vertical" and "horizontal" indices can contain an arbitrary number of observations that can get condensed into one value or kept separate in different columns. There is no relationship between the number of index types, though there must be at least one "vertical" index.

The `'vertical'` and `'horizontal'` settings determine which columns of the dataframes in `score` are included in the n-gram output. `score` is a list of two dataframes, the vertical observations `DataFrame` and the horizontal observations `DataFrame`.

The format of the vertical and horizontal settings is very important and will decide the structure of the resulting n-gram results. Both the vertical and horizontal settings should be a list of tuples. If the optional horizontal setting is passed, its list should be of the same length as that of the vertical setting. Inside of each tuple, enter the column names of the observations that you want to include in each value. For example, if you want to make 3-grams of notes in the tenor in a four-voice choral, use the following settings (NB: there is no horizontal element in this simple query so no horizontal setting is passed. In this scenario you would need to pass the noterest indexer results as the only dataframe in the "score" list of dataframes.):

```
>>> settings = {
        'n': 3,
        'vertical': [('2',)]
    }
```

If you want to look at the 4-grams in the interval pairs between the bass and soprano of a four-voice choral and track the melodic motions of the bass, the `score` argument should be a 2-item list containing the IntervalIndexer results dataframe and the `HorizontalIntervalIndexer` dataframe. Note that the `HorizontalIntervalIndexer` results must have been calculated with the `'horiz_attach_later'` setting set to `True` (this is in order to avoid an indexing nightmare). The settings dictionary to pass to this indexer would be:

```
>>> settings = {
        'n': 4,
        'vertical': [('0,3',)],
        'horizontal': [('3',)]
    }
```

If you want to get 'figured-bass' 2-gram output from this same 4-voice choral, use the same 2-item list for the score argument, and then put all of the voice pairs that sound against the bass in the same tuple in the vertical setting. Here's what the settings should be:

```
>>> settings = {
        'n': 2,
        'vertical': [('0,3', '1,3', '2,3')],
        'horizontal': [('3',)]
    }
```

In the example above, if you wanted stacks of vertical events without the horizontal connecting events, you would just omit the `'horizontal'` setting from the settings dictionary and also only include the vertical observations in the `score` list of dataframes.

If instead you want to look at all the pairs of voices in the 4-voice piece, and always track the melodic motions of the lowest voice in that pair, then put each pair in a different tuple, and in the voice to track melodically in the corresponding tuple in the horizontal list. Since there are 6 pairs of voices in a 4-voice piece, both your vertical and horizontal settings should be a list of six tuples. This will cause the resulting n-gram results dataframe to have six columns of observations. Your settings should look like this:

```
>>> settings = {
        'n': 2, 'vertical': [
            ('0,1',),
            ('0,2',),
            ('0,3',),
            ('1,2',),
            ('1,3',),
            ('2,3')
        ],
        'horizontal': [
            ('1',),
            ('2',),
            ('3',),
            ('2',),
            ('3',),
            ('3',)
        ]
    }
```

Since we often want to look at all the pairs of voices in a piece, you can set the `'vertical'` setting to `'all'` and this will get all the column names from the first dataframe in the score list of dataframes. Similarly, as we often want to always track the melodic motions of the lowest or highest voice in the vertical groups, the horizontal setting can be set to `'highest'` or `'lowest'` to automate this voice selection. This means that the preceeding query can also be accomplished with the following settings:

```
>>> settings = {
        'n': 2,
        'vertical': 'all',
        'horizontal': 'lowest'
    }
```

The `'brackets'` setting will set off all the vertical events at each time point in square brackets '[]' and horizontal observations will appear in parentheses '()'. This is particularly useful if there are multiple observations in each vertical or horizontal slice. For example, if we wanted to redo the query above where n = 4, but this time tracking the melodic motions of both the upper and the lower voice, it would be a good idea to set 'brackets' to `True` to make the results easier to read. The settings would look like this:

```
>>> settings = {
        'n': 4,
        'vertical': [('0,3',)],
        'horizontal': [('0', '3',)],
        'brackets': True
    }
```

If you want n-grams to terminate when finding one or several particular values, you can specify this by passing a list of strings as 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.

Once you've chosen the appropriate settings, to actually run the indexer call it like this:

Example:

```
>>> from vis.models.indexed_piece import Importer
>>> ip = Importer('pathnameToScore.xml')
>>> ngram_settings = {
        'n': 2,
        'vertical': 'all',
        'horizontal': 'lowest'
    }
>>> vert_settings = {
        'quality': 'chromatic',
        'simple or compound': 'simple',
        'directed': True
    }
>>> horiz_settings = {
        'quality': 'diatonic with quality',
        'simple or compound': 'simple',
        'directed': True,
        'horiz_attach_later': True
    }
>>> vert_ints = ip.get_data('vertical_interval', settings=vert_settings)
>>> horiz_ints = ip.get_data('horizontal_interval', settings=horiz_settings)
>>> ip.get_data('ngram', data=[vert_ints, horiz_ints], settings=ngram_settings)
```

`default_settings` = {'open-ended': False, 'vertical': 'all', 'brackets': True, 'continuer': '_', 'terminator': [], 'horizo

`possible_settings` = ['horizontal', 'vertical', 'n', 'open-ended', 'brackets', 'terminator', 'continuer', 'align']

A list of possible settings for the *NGramIndexer*.

Parameters

- **`'horizontal'`** (*list of tuples of strings, default []*) – Selectors for the columns to consider as "horizontal."
- **`'vertical'`** (*list of tuples of strings, default 'all'.*) – Selectors for the column names to consider as "vertical."
- **`'n'`** (*int*) – The number of "vertical" events per n-gram.

- **'open-ended'** (boolean, default `False`.) – Appends the next horizontal observation to n-grams leaving them open-ended.
- **'brackets'** (*bool, default True.*) – Whether to use delimiters around event observations. Square brakets [] are used to set off vertical events and round brackets () are used to set off horizontal events. This is particularly important to leave as `True` (default) for better legibility when there are multiple vertical or multiple horizontal observations at each slice.
- **'terminator'** (*list of str, default []*) – Do not find an n-gram with a vertical item that contains any of these values.
- **'continuer'** (*str, default '_'.*) – 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 new index of the piece in the form of a class:*~pandas.DataFrame* with as many columns as there are tuples in the 'vertical' setting of the passed settings.

noterest Module

Index note and rest objects into pandas DataFrame(s).

class vis.analyzers.indexers.noterest.**MultiStopIndexer**(*score*)
Bases: *vis.analyzers.indexer.Indexer*

Index `Note`, `Rest`, and `Chord` objects in a `DataFrame`.

`Rest` objects become `'Rest'`, and `Note` objects become the string-format version of their `nameWithOctave` attribute. `Chord` objects get unpacked into their constituent pitches.

This indexer is meant to be called indirectly with a call to `get_data` on an indexed piece in the manner of the following example.

Example:

```
>>> from vis.models.indexed_piece import Importer
>>> ip = Importer('path_to_piece.xml')
>>> ip.get_data('multistop')
```

required_score_type = 'pandas.DataFrame'

run()
Make a new index of the note and rest names in the piece. When a single part has chord objects, those chords get separated out into as many columns as there are notes in the chord with the greatest number of notes. This means that there can be more columns in this dataframe than there are parts in the piece.

Returns A `DataFrame` of the new indices. The columns have a `MultiIndex`.

Return type `pandas.DataFrame`

class vis.analyzers.indexers.noterest.**NoteRestIndexer**(*score*)
Bases: *vis.analyzers.indexer.Indexer*

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

`Rest` objects become `'Rest'`, and `Note` objects become the string-format version of their `nameWithOctave` attribute.

This indexer is meant to be called indirectly with a call to `get_data` on an indexed piece in the manner of the following example.

Example:

```
>>> from vis.models.indexed_piece import Importer
>>> ip = Importer('path_to_piece.xml')
>>> ip.get_data('noterest')
```

required_score_type = 'pandas.DataFrame'

vis.analyzers.indexers.noterest.**multistop_ind_func**(*event*)

Used internally by *MultiStopIndexer*. Convert `Note` and `Rest` objects into a string and convert the `Chord` objects into a list of the strings of their consituent pitch objects. The results must be contained in a tuple or a list so that chords can later be unpacked into different 1-voice strands.

Parameters **event** (*A music21 note, rest, or chord object.*) – A music21 note, rest, or chord object which get queried for their names.

Returns A one-tuple containing a string representation of the note or rest, or if the event is a chord, a list of the strings of the names of its constituent pitches.

Return type 1-tuple of str or list of strings

Examples:

```
>>> from noterest.py import indexer_func
>>> from music21 import note,
>>> indexer_func(note.Note('C4'))
(u'C4',)
>>> indexer_func(note.Rest())
(u'Rest',)
>>> indexer_func(chord.Chord([note.Note('C4'), note.Note('E5')]))
[u'C4', u'E5']
```

vis.analyzers.indexers.noterest.**noterest_ind_func**(*event*)

Used internally by *NoteRestIndexer*. Convert `Note`, `Rest`, and `If you want to keep all the pitches in chords, consider using the :class:`MultiStopIndexer` instead.

Parameters **event** (*A music21 note, rest, or chord object.*) – A music21 note, rest, or chord object which get queried for their names.

Returns A one-tuple containing a string representation of the note or rest, or if the event is a chord, a list of the strings of the names of its constituent pitches.

Return type 1-tuple of str or list of strings

Examples:

```
>>> from noterest.py import indexer_func
>>> from music21 import note,
>>> indexer_func(note.Note('C4'))
u'C4'
>>> indexer_func(note.Rest())
u'Rest'
>>> indexer_func(chord.Chord([note.Note('E5'), note.Note('C4')]))
u'E5'
```

vis.analyzers.indexers.noterest.**unpack_chords**(*df*)

The c in nrc in methods like _get_m21_nrc_objs() stands for chord. This method unpacks music21 chords into a list of their constituent pitch objects. These pitch objects can be queried for their `nameWithOctave` in the same way that note objects can in music21. This works by broadcasting the list of pitches in each chord object in each part's elements to a dataframe of note, pitch, and rest objects. So each part that had chord objects in it gets represented as a dataframe instead of just a series. Then the series from the parts that didn't have chords in them get concatenated with the parts that did, resulting in potentially more columns in the final dataframe then there are parts in the score.

`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 meter. 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!

Concerning the "dynamic-offset method", this can be accessed by passing the string "dynamic" for the quarterLength setting. This type of analysis is still experimental and comes with no guarantee that it will work accurately. It has the important known limitation that it only applies to Renaissance polyphony in which the contrapuntal rhythm is only ever in duple groupings. For more on contrapuntal rhythm, see:

DeFord, Ruth. Tactus Mensuration, and Rhythm in Renaissance Music. Cambridge: Cambridge University Press, 2015.

For a more thorough explanation of the experimental dynamic-offset method see (especially chapter 4):

Morgan, Alexander. "Renaissance Interval-Succession Theory: Treatises and Analysis." PhD diss., McGill University, 2017.

Helper functions have been implemented to facilitate the use of the dynamic-offset method, so you can run analyses in the following way:

from vis.models.indexed_piece import Importer ip = Importer('full_path_to_piece_in_symbolic_notation.xml') # assuming you want to apply the offset filter to the noterest # indexer results: nr = ip.get_data('noterest') setts = {'quarterLength': 'dynamic'} filtered_nr = ip.get_data('offset', data=nr, settings=setts)

`default_settings` = {'dom_data': [], 'method': 'ffill', 'mp': True}

`possible_settings` = ['quarterLength', 'dom_data', 'method', 'mp']
A `list` of possible settings for the *`FilterByOffsetIndexer`*.

Parameters

- **`'quarterLength'`** (*`float or string`*) – 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). For dynamic (i.e. variable) and context-dependent value, pass the string 'dynamic'.

- **'dom_data'** – A list of DataFrames and one integer is required here if the 'quarterLength' setting is set to 'dynamic'. This list should contain the dissonance, duration, beatstrength, and noterest indexer dataframes and finally the "highest_time" of the piece or movement in that order. The correct information is automatically fetched if this indexer is called on an IndexedPiece object via the get_data method() if the 'data' argument in that method is not passed.
- **'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.
- **'mp'** (*boolean*) – Multiprocesses when True (default) or processes serially when False.

Examples:

```
>>> from vis.models.indexed_piece import Importer
>>> ip = Importer('path_to_piece.xml')
>>> notes = ip.get_data('noterest')
>>> setts = {'quarterLength': 2}
>>> ip.get_data('offset', data=notes, settings=setts)
```

Note that other analysis results can be passed to the offset # indexer too, such as the IntervalIndexer results as in the # following example. Also, the original column names (or names of # the series if a list of series was passed) are retained, though # the highest level of the columnar multi-index gets overwritten

```
>>> from vis.models.indexed_piece import Importer
>>> ip = Importer('path_to_piece.xml')
>>> intervals = ip.get_data('vertical_interval')
>>> setts = {'quarterLength': 2}
>>> ip.get_data('offset', data=intervals, settings=setts)
```

required_score_type = 'pandas.Series'
The *FilterByOffsetIndexer* uses `pandas.Series` objects.

run()
Regularize the observed offsets for the Series input.

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`

over_bass Module

class vis.analyzers.indexers.over_bass.**OverBassIndexer**(*score*, *settings=None*)
Bases: *vis.analyzers.indexer.Indexer*

Using horizontal events and vertical intervals, this finds the intervals over the bass motion.

Call this indexer via the `get_data()` method of either an `indexed_piece` object or an `aggregated_pieces` object (see examples below). The DataFrames passed in the 'score' argument should be concatenated

Parameters

- **'horizontal'** (*int*) – The horizontal voice you wish to use as a bass. If not indicated, this is automatically assigned to the lowest voice.
- **'type'** (*str*) – The type of horizontal event you wish to index. The default is 'notes', which should be used if you are passing in the results of the `NoteRestIndexer`. If you are passing in the results of the `HorizontalIntervalIndexer`.

Example:

```
>>> from vis.models.indexed_piece import Importer
>>> ip = Importer('path_to_piece.xml')
```

This example provides note names for the tracked voice (usually the bass voice):

```
>>> input_dfs = [ip.get_data('noterest'),
        ip.get_data('vertical_interval')]
>>> ob_setts = {'type': 'notes'}
>>> ip.get_data('over_bass', data=input_dfs, settings=ob_setts)
```

To use the OverBassIndexer with the melodic intervals of the tracked voice instead, do this:

```
>>> input_dfs = [ip.get_data('horizontal_interval'),
        ip.get_data('vertical_interval')]
>>> ob_setts = {'type': 'intervals'}
>>> ip.get_data('over_bass', data=input_dfs, settings=ob_setts)
```

possible_settings = ['horizontal', 'type']

required_score_type = 'pandas.DataFrame'

run()
Make a new index of the intervals over the bass motion.

Returns A `DataFrame` of the intervals over the bass.

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.

Example:

Prepare an indexed piece:

```
>>> from vis.models.indexed_piece import Importer
>>> ip = Importer('path_to_piece.xml')
```

This example filters the repeats out of the `NoteRestIndexer` results, but any can be passed:

```
>>> notes = ip.get_data('noterest')
>>> ip.get_data('repeat', data=notes)
```

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 = ['fake_setting']
This is a list of string 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 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 str

windexer Module

class vis.analyzers.indexers.windexer.**Windexer**(*score*, *settings=None*)
Bases: *vis.analyzers.indexer.Indexer*

Indexer that creates new a new windowed version of other indexer results.

Parameters **'window_size'** (*integer*) – The size of the window of the DataFrame that you would like to look at. The default setting is 4.

Example:

Prepare an indexed piece:

```
>>> from vis.models.indexed_piece import Importer
>>> ip = Importer('path_to_piece.xml')
```

This example prepares "windows" of notes and rests, but any dataframe could be used:

```
>>> notes = ip.get_data('noterest')
>>> setts = {'window_size': 4}
>>> ip.get_data('windexer', data=notes, settings=setts)
```

default_settings = {'window_size': 4}

possible_settings = ['window_size']

required_score_type = 'pandas.DataFrame'

run()
Make a new windowed index of the indexer results.

Returns The new windowed DataFrame.

Return type pandas.DataFrame

models Package

models Package

aggregated_pieces Module

The model representing data from multiple *IndexedPiece* instances.

class vis.models.aggregated_pieces.**AggregatedPieces**(*pieces=None*, *metafile=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**(*ind_analyzer=None*, *combined_experimenter=None*, *settings=None*, *data=None*)
Get the results of an Indexer or an Experimenter run on all the IndexedPiece objects either individually, or all together. If settings are provided, the same settings dict will be used throughout.

In VIS, analyzers are broken down into two categories: Indexers which associate observations with a specific moment in a piece, and Experimenters which still work with musical observations, but do not associate them with a specific moment in a specific IndexedPiece. For example, the noterest.NoteRestIndexer associates each note and rest with a time point in a given IndexedPiece, but if we then use the frequency.FrequencyExperimenter to count the number of times each type of note or rest happens, these counts will not and cannot be associated with a specific time point.

All VIS Indexers and most Experimenters run on each piece individually, and so if these results are desired, the analyzer in question should be assigned to the ind_analyzer argument. The barchart.RBarChart and aggregator.ColumnAggregator experimenters often combine the data of several pieces together. The

frequency.FrequencyExperimenter can also be used this way. If this is the desired behavior, supply the appropriate Experimenter as the combined_experimenter argument.

Examples

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

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

Parameters

- **ind_analyzer** (*str or VIS Indexer or Experimenter class.*) – The analyzer to run.
- **settings** (*dict*) – Settings to be used with the analyzer. Only use if necessary.
- **data** (Depends on the requirement of the analyzer designated by the `analyzer_cls` argument. Usually a list of `pandas.DataFrame`.) – Input data for the analyzer to run. If this is provided for an indexer that normally caches its results (such as the NoteRestIndexer, the DurationIndexer, etc.), the results will not be cached since it is uncertain if the input passed in the `data` argument was calculated on this indexed_piece.

Returns Results of the analyzer.

Return type Depending on the `analyzer_cls`, either a `pandas.DataFrame` or more often a list of `pandas.DataFrame`.

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 :param field: The name of the field to be accessed or modified. :type field: str :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 str.

`indexed_piece` Module

This model represents an indexed and analyzed piece of music.

`vis.models.indexed_piece.`**`Importer`**(*location*, *metafile=None*)

Import the file, website link, or directory of files designated by `location` to music21 format.

Parameters **location** (*str*) – Location of the file to import on the local disk.

Returns An *IndexedPiece* or an AggregatedPieces object if the file passed imports as a music21.stream.Score or music21.stream.Opus object respectively.

Return type A new *IndexedPiece* or AggregatedPieces object.

class vis.models.indexed_piece.**IndexedPiece**(*pathname=''*, *opus_id=None*, *score=None*, *metafile=None*, *username=None*, *password=None*)

Bases: object

Hold indexed data from a musical score, and the score itself. IndexedPiece objects are VIS's basic representations of a piece of music and also a container for metadata and analyses about that piece. An IndexedPiece object should be created by passing the pathname of a symbolic notation file to the Importer() method in this file. The Importer() will return an IndexedPiece object as long as the piece did not import as an opus. In this case Importer() will return an AggregatedPieces object. Information about an IndexedPiece object from an indexer or an experimenter should be requested via the get_data() method. If you want to access the full music21 score object of a VIS IndexedPiece object, access the _score attribute of the IndexedPiece object. See the examples below:

Examples # Creat an IndexedPiece object from vis.models.indexed_piece import Importer ip = Importer('path_to_file.xml')

Get the results of an indexer or experimenter (noterest and dissonance indexers shown) noterest_results = ip.get_data('noterest') dissonance_results = ip.get_data('dissonance')

Access the full music21 score object of the file ip._score

get_data(*analyzer_cls*, *data=None*, *settings=None*)

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

Parameters

- **analyzer_cls** (*str or VIS Indexer or Experimenter class.*) – The analyzer to run.
- **settings** (*dict*) – Settings to be used with the analyzer. Only use if necessary.
- **data** (Depends on the requirement of the analyzer designated by the analyzer_cls argument. Usually a pandas.DataFrame or a list of pandas.Series.) – Input data for the analyzer to run. If this is provided for an indexer that normally caches its results (such as the NoteRestIndexer, the DurationIndexer, etc.), the results will not be cached since it is uncertain if the input passed in the data argument was calculated on this indexed_piece.

Returns Results of the analyzer.

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

Raises RuntimeWarning 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.

load_url(*url*)

measure_index(*dataframe*)

Multi-indexes the index of the passed dataframe by adding the measures to the offsets. The passed dataframe should be of an indexer's results, not an experimenters. Also adds index labels. Note that this method currently does not work with midi files, because VIS cannot detect measures in midi files since they are not encoded in midi. Also note that this method should ideally only be used at the end of

a set of analysis steps, because there is no guarantee that the resultant multi-indexed dataframe will not cause problems if passed to a subsequent indexer.

Example from vis.models.indexed_piece import Importer() # Make an IndexedPiece object out of a symbolic notation file: ip = Importer('path_to_file.xml') # Get some results from an indexer (not an experimenter): df = ip.get_data('horizontal_interval') # Multi-index the dataframe index by adding the measure informaiton: ip.measure_index(df)

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** (*str*) – 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 string)

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

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', ...]

vis.models.indexed_piece.**auth_get** (*url*, *csrftoken*, *sessionid*)

Use a csrftoken and sessionid to request a url on the elvisdatabase.

vis.models.indexed_piece.**login_edb**(*username*, *password*)
Return csrf and session tokens for a login.

workflow Module

Deprecated since version 3.0.0: The WorkflowManager is deprecated as of VIS 3.0.0 and will be entirely removed in VIS 4.0. It was an important part of VIS in earlier versions but the iterative caching strategy implemented in VIS 3.0 obviates the need for the WorkflowManager and so it is being phased out for simplicity. Most of its functionality still works with VIS 3.0, however, it is no longer being maintained or supported.

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

Warning: The WorkflowManager is deprecated as of VIS 3.0 and will be entirely removed in VIS 4.0. Most of its functionality still works with VIS 3.0 but this is not guaranteed and it is no longer being supported in development.

Parameters **pathnames** (list or tuple of string 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 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** (*str*) – The type of data to load. Defaults to `'pieces'`.
- **pathname** (*str*) – 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 *output()*.
- `'stata'` to load data from a previous *output()*.
- `'pickle'` to load data from a previous *output()*.

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 `len()` of this *WorkflowManager*.
- **field** (*str*) – The name of the field to be accessed or modified.
- **value** (*object*) – 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

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

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`.

Note: If `count frequency` is set to `False` for CSV, Stata, Excel, or HTML output, the `top_x` and `threshold` parameters are ignored.

Parameters

- **instruction** (*str*) – The type of visualization to output.

- **pathname** (*str*) – 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. The other formats will return a list if the `count frequency` setting is `False`.

Return type str or [str]

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 th** 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** (*str*) – 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`** (*`str`*) – 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`. This 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 `str` 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** (*str*) – 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)
```

Indices and Tables

- genindex
- modindex
- search

Python Module Index

v

Index

G

H

I

L

M

N

O

P

R

S

T

U

V

W