PyTuning Documentation

Release 0.7.1

Mark Conway Wirt

Sep 02, 2023

Contents

PyTuning is a Python library intended for the exploration of musical scales and microtonalities. It can be used by developers who need ways of calculating, analyzing, and manipulating musical scales, but it can also be used interactively.

It makes heavy use of the SymPy package, a pure-Python computer algebra system, which allows scales and scale degrees to be manipulated symbolically, with no loss of precision. There is also an optional dependency on Matplotlib (and Seaborn) for some visualizations that have been included in the package.

Some of the package's features include:

- Creation of scales in a variety of ways (EDO, Euler-Fokker, Diatonic, Harmonic, from generator intervals, etc.)
- Ability to represent created scales in ways that are understood by external software (Scala, Timidity, Fluidsynth, Yoshimi, Zynaddsubfx).
- Some analysis functions (for example, PyTuning provides a framework for searching for scale modes based upon defined metric functions and combinatorial analysis). Also included are some number-theoretic functions, such as prime limits and odd limits.
- Some scale visualizations.
- Interactive use.

As a simple example, to create a 31-TET scale and then create a tuning table for the timidity soft-synth:

```
scale = create_edo_scale(31)
tuning_table = create_timidity_tuning(scale, reference_note=69)
```

The design of PyTuning is purposefully simple so that non-computer professionals can use it without much difficultly (musicians, musicologist, interested people of all stripes).

In scope this project is similar to the Scala software package, with a few differences:

- Scala is a mature, full-featured package that includes many, many scales and functions for manipulating and analyzing those scales. This project is much newer and less mature; its scope is currently much less (but hopefully it will be easy to extend).
- PyTuning is written in Python and relies on modern, well maintained dependencies. Scala is written in Ada, and while this is an interesting choice, it probably limits the population of users who *could* change or extend it should a need arise.
- Scala is mainly an application. PyTuning is a development library, but with ways for non-programmers to use it interactively.
- This package does *not* interact with sound cards or audio drivers, so one can't play a scale directly. There are, however, functions for exporting scales into other software packages so that music and sound can be produced.

CHAPTER 1

Installation

PyTuning runs under Python 2.7.X and 3.X.

The easiest way to install PyTuning is via the Python Package Index, with which Pytuning is registered:

```
pip install pytuning
```

There are two hard dependencies for PyTuning: SymPy and NumPy. SymPy is a pure Python library and `pip` will handle its installation nicely. NumPy is a more complicated package and if installed via `pip` may involve much compilation; it would probably behoove you to install the package manually via whatever mechanism your platform provides before `pip` installing the package .

If you are running the package interactively it is recommended that the Jupyter interactive shell be installed. This is discussed in the documentation under the notes on Interactive use.

The source-code is available on GitHub, where it can be cloned and installed.

CHAPTER 2

Documentation

Documentation for the package can be found on Read the Docs. Documentation for the latest development version is here.

CHAPTER 3

Roadmap

More scales, more visualizations, more analysis functions. Pull requests are welcome!

3.1 Basic Concepts

PyTuning is purposefully designed to be as simple as possible, so that non-programmers (musicologists, musicians, etc.) can use it without too much difficulty: the data structures are relatively simple; there are currently no classes defined, instead opting for an imperative/procedural approach.

Regardless of how it is used (interactively or as a development library), the user should have a good understanding of some of the basic, foundational concepts the package uses.

3.1.1 Scales

A **scale** is, simply, a list of degrees. By convention the list is bookended by the unison and the octave, with each degree given as a frequency ratio relative to the root tone of the scale. The first degree is always 1, the ratio of the first degree to the first degree. Most commonly the last degree is 2, as the octave is usually twice the frequency of the root (although the package has some support for non-standard "octaves").

As an example, this is the standard 12-tone equal temperament scale (sometimes referred to as 12-TET, or 12-EDO, for Equal Division of the Octave).

$$\left[1, \quad \sqrt[12]{2}, \quad \sqrt[6]{2}, \quad \sqrt[4]{2}, \quad \sqrt[3]{2}, \quad 2^{\frac{5}{12}}, \quad \sqrt{2}, \quad 2^{\frac{7}{12}}, \quad 2^{\frac{2}{3}}, \quad 2^{\frac{3}{4}}, \quad 2^{\frac{5}{6}}, \quad 2^{\frac{11}{12}}, \quad 2\right]$$

A few things to note:

- As mentioned previously, the scale includes the unison and octave.
- Each scale degree is a SymPy number, so it is represented symbolically. Note that algebraic simplifications are performed by default.
- Even though the length of this list of 13, it is considered a 12 note scale, because the unison and the octave is in actuality the same note. Many of the functions in this package ask one to choose the number of notes or degrees to be used in this function. For these you should follow this convention.

(For those who are curious: the generation of scales is documented in *Scale Creation*, but the above scale was generated with the following code:

```
from pytuning.scales import create_edo_scale
edo_12_scale = create_edo_scale(12)
```

Simplified versions of most functions are provided in the interactive environment.)

3.1.2 Degrees

Scale degrees (which are expressed as frequency rations relative to the tonic of the scale) are expressed in SymPy values. In practical terms the `Integer` and `Rational` class will be used the most, but SymPy is a full-featured package, and you may benefit from having some familiarity with it.

An example of a few degrees:

```
import sympy as sp

unison           = sp.Integer(1)                      # Normal unison
octave           = sp.Integer(2)                      # Normal octave
perfect_fifth    = sp.Rational(3,2)                   # As a rational number
minor_second_tet = sp.Integer(2) ** sp.Rational(1,12) # the 12th root of 2
                                                      # could also by sp.root(2,12)
lucy_L           = sp.root(2,2*sp.pi)                 # Lucy scale Long step
```

Will yield the following:

$$1$$
$$2$$
$$\frac{3}{2}$$
$$\sqrt[12]{2}$$
$$2^{\frac{1}{2\pi}}$$

SymPy manipulates all values analytically, but sometimes one needs a floating approximation to a degree (for example, tuning a synthesizer usually needs frequencies expressed as floating point numbers). For the `evalf()` member function can be used:

```
print(unison.evalf())
1.00000000000000
print(octave.evalf())
2.00000000000000
print(perfect_fifth.evalf())
1.50000000000000
print(minor_second_tet.evalf())
1.05946309435930
print(lucy_L.evalf())
1.11663288009114
```

3.1.3 Modes

A **mode** is a selection of notes from a scale, and is itself a list of degrees (and therefore is also a scale). A mode can be produced from a scale by applying a **mask** to the scale. Again, the functions involved are documented elsewhere, but as example this is how we would produce the standard major scale (which in the context of this package would be referred to as a mode):

```
major_mask = (0,2,4,5,7,9,11,12)
major_mode = mask_scale(edo_12_scale, major_mask)
```

which produces the following scale:

$$\left[1, \quad \sqrt[6]{2}, \quad \sqrt[3]{2}, \quad 2^{\frac{5}{12}}, \quad 2^{\frac{7}{12}}, \quad 2^{\frac{3}{4}}, \quad 2^{\frac{11}{12}}, \quad 2\right]$$

3.1.4 Mode Objects

Some functions in this package return a **mode object**. For example, the `find_best_modes()` function will take a scale and find a mode (or modes), based upon some consonance metric function. Here is one such object, which is implemented as a Python `dict`.

```
{'mask': (0, 2, 3, 5, 7, 9, 10, 12),
 'metric_3': 22.1402597402597,
 'original_scale': [1,
  256/243,
  9/8,
  32/27,
  81/64,
  4/3,
  1024/729,
  3/2,
  128/81,
  27/16,
  16/9,
  243/128,
  2],
 'scale': [1, 9/8, 32/27, 4/3, 3/2, 27/16, 16/9, 2],
 'steps': [2, 1, 2, 2, 2, 1, 2],
 'sum_distinct_intervals': 12,
 'sum_p_q': 161,
 'sum_p_q_for_all_intervals': 4374,
 'sum_q_for_all_intervals': 1822}
```

The meaning of these keys:

- `original_scale` is the original scale which was input into the function. In this example is was a Pythagorean scale.
- `scale` is the output scale of the function
- `mask` is the mask of the original scale that produces the output
- `steps` is similar to mask, but reported in a different format. Each entry in the steps list represents the number of degrees in the original scale between successive degrees in the returned scale. The standard major scale, for example, would be represented by $[2, 2, 1, 2, 2, 2, 1]$.

In this example there are also other keys included. `sum_distinct_intervals`, `sum_p_q`, `sum_p_q_for_all_intervals`, `sum_q_for_all_intervals`, and `metric_3` are the outputs of calculated metric functions. This particular mode, for example, has a rating of 161 by the `sum_p_q` metric.

Metric functions are describe briefly below, and in more detail in *Metric Functions*.

3.1.5 Tuning Tables

Tuning Tables are a representation of a scale, usually a string (which can be written to a file), which can be understood by an external software package. As an example, to take a standard Pythagorean scale and produce a representation understood by Scala:

```
pythag_scale       = create_pythagorean_scale()
scala_tuning_table = create_scala_tuning(pythag_scale, "Pythagorean Scale")
```

The variable `scala_tuning_table` now contains the following:

```
! Scale produced by pytuning. For tuning yoshimi or zynaddsubfx,
! only include the portion below the final '!'
!
Pythagorean Scale
 12
!
256/243
9/8
32/27
81/64
4/3
1024/729
3/2
128/81
27/16
16/9
243/128
2/1
```

For many tuning tables one has to pin the scale to some reference frequency. For this the convention of MIDI note number is employed. For example, in the MIDI standard the note `69` is A 440 Hz, so by specifying a reference of 69, the corresponding entry in the table would be 400 Hz, and this would represent the root or tonic degree of the scale.

Exporting the above scale in a Csound compatible format:

```
csound_tuning_table = create_csound_tuning(pythag_scale, reference_note=69)
```

yields the following:

```
f1 0 256 -2     8.14815     8.70117     9.16667     9.65706    10.31250    10.86420  ␣
↪ 11.60156    12.22222 \
              13.05176    13.75000    14.48560    15.46875    16.29630    17.40234  ␣
↪ 18.33333    19.31413 \
              20.62500    21.72840    23.20313    24.44444    26.10352    27.50000  ␣
↪ 28.97119    30.93750 \
              32.59259    34.80469    36.66667    38.62826    41.25000    43.45679  ␣
↪ 46.40625    48.88889 \
              52.20703    55.00000    57.94239    61.87500    65.18519    69.60938  ␣
↪ 73.33333    77.25652 \
              82.50000    86.91358    92.81250    97.77778   104.41406   110.00000  ␣
↪115.88477   123.75000 \
             130.37037   139.21875   146.66667   154.51303   165.00000   173.82716  ␣
↪185.62500   195.55556 \
             208.82813   220.00000   231.76955   247.50000   260.74074   278.43750  ␣
↪293.33333   309.02606 \
             330.00000   347.65432   371.25000   391.11111   417.65625   440.00000  ␣
↪463.53909   495.00000 \
             521.48148   556.87500   586.66667   618.05213   660.00000   695.30864  ␣
↪742.50000   782.22222 \
             835.31250   880.00000   927.07819   990.00000  1042.96296  1113.75000  ␣
↪1173.33333  1236.10425 \
```

(continues on next page)

(continued from previous page)

```
             1320.00000  1390.61728  1485.00000  1564.44444  1670.62500  1760.00000 ␣
↪1854.15638  1980.00000 \
             2085.92593  2227.50000  2346.66667  2472.20850  2640.00000  2781.23457 ␣
↪2970.00000  3128.88889 \
             3341.25000  3520.00000  3708.31276  3960.00000  4171.85185  4455.00000 ␣
↪4693.33333  4944.41701 \
             5280.00000  5562.46914  5940.00000  6257.77778  6682.50000  7040.00000 ␣
↪7416.62551  7920.00000 \
             8343.70370  8910.00000  9386.66667  9888.83402 10560.00000 11124.93827␣
↪11880.00000 12515.55556
```

This is a 128-entry table, mapping note number to absolute frequency. Csound's `table` opcode can be used to index into the table and play the appropriate frequency, using something like the following:

```
inote     init          p4
iveloc    init          p5
ifreq     table         inote, 1
a1        oscil         iveloc, ifreq, 2
          outs          a1, a1
```

(This assumes that p4 in the orchestra file contains MIDI note numbers, of course. If you use a different convention there are translation opcodes that can be used.)

3.1.6 Metric Functions

Metric Functions are functions that takes a scale as an input and returns a numeric value calculated from that scale. It is used, for example, in `find_best_modes()` to evaluate the consonance of a scale (`find_best_modes()` uses a metric to evaluate the consonance of all possible modes of a scale and returns the evaluation of those modes as a **mode_object**).

The return value of a metric function should be a `dict` with a unique string identifier as the key and the metric as the value.

As an example, the following is one of the package-defined metrics:

`pytuning.metrics.`**`sum_p_q_for_all_intervals`**`(`*scale*`)`

Calculate a metric for a scale

Parameters **`scale`** – The scale (i.e., a list of `sympy.Rational` values)

Returns The metric.

This metric is an estimate of scale consonance. It is formed by examining all unique intervals in the scale, and creating a numeric value based upon the summation of the numerators and denominators for all those intervals.

While the metric is numerically defined for ratios expressed as irrational or transcendental numbers, it is really only meaningful for scales with just degrees (ratios expressed as rational numbers).

Smaller values are more consonant.

As an example of use, the following:

```
pythag = create_pythagorean_scale()
metric = sum_p_q_for_all_intervals(pythag)
```

yields the following:

```
{'sum_p_q_for_all_intervals': 1092732}
```

3.2 Scale Creation

There are several scale-creation functions in the package. They are found in `pytuning.scales` and can be imported into the program's namespace with

```
from pytuning.scales import *
```

(Note that for interactive use these are imported by default).

3.2.1 The Harmonic Scale

We'll start with the harmonic scale; it will illustrate many of the concepts used in scale creation.

There are two important concepts to understand:

- Normalization: If a scale is normalized (which is the default in all cases), then the intervals of the scale are normalized to fall within a single octave. This means scaling the interval either up or down the number of octaves needed to make the interval fall between the unison and the octave.
- The Octave: Normally an octave is defined as a doubling of frequency (2), but it is possible to define an octave by some other number. If this is the case the normalization will takes place over this new octave.

The function to create a harmonic scale is, `create_harmonic_scale`:

`pytuning.scales.`**`create_harmonic_scale`**(*first_harmonic*, *last_harmonic*, *normalize=True*, *octave=2*)

Create a harmonic scale

Parameters

- **`first_harmonic`** – The first harmonic
- **`last_harmonic`** – The last harmonic
- **`normalize`** – If true, normalize the scale to an octave (2/1 by default, otherwise taken from `octave`)
- **`octave`** – The definition of the formal octave.

Returns The scale

As an example of use, a normalized scale constructed from harmonics 3 to 20:

```
scale = create_harmonic_scale(3,20)
```

which yields:

$$\left[1, \frac{13}{12}, \frac{7}{6}, \frac{5}{4}, \frac{4}{3}, \frac{17}{12}, \frac{3}{2}, \frac{19}{12}, \frac{5}{3}, \frac{11}{6}, 2\right]$$

To create a non-normalized scale:

```
scale = create_harmonic_scale(3,10, normalize=False)
```

which yields:

$$\left[1, \frac{4}{3}, \frac{5}{3}, 2, \frac{7}{3}, \frac{8}{3}, 3, \frac{10}{3}\right]$$

As an example, if we create a non-normalized harmonic scale of 10 harmonics:

```
harmonic_scale = create_harmonic_scale(1, 10, normalize=False)
```

We have the following scale:

$$[1, \quad 2, \quad 3, \quad 4, \quad 5, \quad 6, \quad 7, \quad 8, \quad 9, \quad 10]$$

If we normalize it each interval is scaled by a power of two to fall within 1 and 2. So, for example, the 9 becomes $\frac{9}{8}$, because the nine must be scaled by three octaves to fall within that range:

$$\frac{9}{8} = \frac{9}{2^3}$$

So the normalized scale is:

$$\left[1, \quad \frac{9}{8}, \quad \frac{5}{4}, \quad \frac{3}{2}, \quad \frac{7}{4}, \quad 2\right]$$

But if we change our octave definition to be 3, we normalize on powers of 3:

```
harmonic_scale = create_harmonic_scale(1, 10, octave=3)
```

yields:

$$\left[1, \quad \frac{10}{9}, \quad \frac{4}{3}, \quad \frac{5}{3}, \quad 2, \quad \frac{7}{3}, \quad \frac{8}{3}, \quad 3\right]$$

3.2.2 Equal Divsion of the Octave (Equal Temprament)

Equal temperament scales can be created with the `create_edo_scale()` function. Note that this function does *not* accept a `normalize` argument, because EDO scales are normalized by definition. If does, however, allow you to change the definition of the formal octave.

`pytuning.scales.`**`create_edo_scale`**`(`*number_tones*, *octave=2*`)`
Create an equal division of octave (EDO, ET) scale.

Parameters

- **`number_tones`** – The number of tones/divisions in the scale
- **`octave`** – The formal octave (frequency ratio)

Example, 12T-ET:

```
edo_scale = create_edo_scale(12)
```

will yield the normal equal-tempered scale used in western music:

$$\left[1, \sqrt[12]{2}, \sqrt[6]{2}, \sqrt[4]{2}, \sqrt[3]{2}, 2^{\frac{5}{12}}, \sqrt{2}, 2^{\frac{7}{12}}, 2^{\frac{2}{3}}, 2^{\frac{3}{4}}, 2^{\frac{5}{6}}, 2^{\frac{11}{12}}, 2\right]$$

Note that the length of the scale is 13, as both the unison and octave are included by convention.

It is also possible to have a non-2 formal octave. The code:

```
edo_scale = create_edo_scale(12,3)
```

will yield:

$$\left[1, \sqrt[12]{3}, \sqrt[6]{3}, \sqrt[4]{3}, \sqrt[3]{3}, 3^{\frac{5}{12}}, \sqrt{3}, 3^{\frac{7}{12}}, 3^{\frac{2}{3}}, 3^{\frac{3}{4}}, 3^{\frac{5}{6}}, 3^{\frac{11}{12}}, 3\right]$$

3.2.3 Scales from a Generator Interval

The `create_equal_interval_scale()` function will generate a scale from a generator interval. This is the base function for several other scale types (for example, the Pythagorean scale is created with a generator interval of $\frac{3}{2}$).

In he creation of a scale, the generator interval can either be used directly (for, for example, making each successive tone a generator interval above the previous tone), or in an inverted sense (making each interval a generator *down* from the previous). This function starts from the unison and walks down the number specified, walking up for the rest of the intervals.

`pytuning.scales.`**`create_equal_interval_scale`**`(`*generator_interval*, *scale_size=12*, *number_down_intervals=6*, *epsilon=None*, *sort=True*, *octave=2*, *remove_duplicates=True*, *normalize=True*`)`

Create a scale with equal-interval tuning

Parameters

- **`generator_interval`** – The interval to use for generation (`sympy` value)
- **`scale_size`** – The number of degrees in the scale
- **`number_down_intervals`** – The number of inverted intervals to use in scale construction.
- **`epsilon`** – Rounding parameter. If set to `None` no rounding is done. Otherwise the scale degrees are rounded to the nearest epsilon
- **`sort`** – If `True`, sort the output by degree size
- **`octave`** – The formal octave
- **`remove_duplicates`** – If `True` remove duplicate entries
- **`normalize`** – IF `True`, normalize the degrees to the octave

In general one should keep epsilon at `None` and perform and rounding outside the function.

This is a base function from which several other scales are derived, including:

- **The Pythagorean scale** A scale with a perfect fifth (3/2) as the generating interval

$$P_5 = \frac{3}{2}$$

- **The quarter-comma meantone scale** A scale in which the generating interval is a perfect fifth narrowed by one quarter of syntonic comma

$$P_5 = \frac{\frac{3}{2}}{\sqrt[4]{\frac{81}{80}}}$$

- **EDO Scales** EDO scales can be generated from an appropriate selection of the fifth. For example, the 12-TET scale would use the fifth:

$$P_5 = \sqrt[\frac{7}{12}]{2}$$

3.2.4 The Pythagorean Scale

This is the standard Pythagorean scale. Note that we can choose the number of up and down intervals in the scale. The default yields the standard scale, with the fourth degree as a diminished fifth, as opposed to the augmented fourth.

`pytuning.scales.`**`create_pythagorean_scale`**(*scale_size=12, number_down_fifths=6, epsilon=None, sort=True, octave=2, remove_duplicates=True*)

Create a Pythagorean scale

Parameters

- **`scale_size`** – The number of degrees in the scale
- **`number_down_fifths`** – The number of inverted fifths to use in scale construction.
- **`epsilon`** – Rounding parameter. If set to `None` no rounding is done. Otherwise the scale degrees are rounded to the nearest epsilon
- **`sort`** – If `True`, sort the output by degree size
- **`octave`** – The formal octave
- **`remove_duplicates`** – If `True` remove duplicate entries

The Pythagorean scale is an even-interval scale with the following generating interval:

$$P_5 = \frac{3}{2}$$

So, for the standard scale we can use:

```
scale = create_pythagorean_scale()
```

yielding:

$$\left[1, \frac{256}{243}, \frac{9}{8}, \frac{32}{27}, \frac{81}{64}, \frac{4}{3}, \frac{1024}{729}, \frac{3}{2}, \frac{128}{81}, \frac{27}{16}, \frac{16}{9}, \frac{243}{128}, 2\right]$$

If we wanted the augmented fourth:

```
scale = create_pythagorean_scale(number_down_fifths=5)
```

yielding:

$$\left[1, \frac{256}{243}, \frac{9}{8}, \frac{32}{27}, \frac{81}{64}, \frac{4}{3}, \frac{729}{512}, \frac{3}{2}, \frac{128}{81}, \frac{27}{16}, \frac{16}{9}, \frac{243}{128}, 2\right]$$

3.2.5 The Quarter-Comma Meantone Scale

`pytuning.scales.`**`create_quarter_comma_meantone_scale`**(*scale_size=12, number_down_fifths=6, epsilon=None, sort=True, octave=2, remove_duplicates=True*)

Create a quarter-comma meantone scale

Parameters

- **scale_size** – The number of degrees in the scale
- **number_down_fifths** – The number of inverted fifths to use in scale construction.
- **epsilon** – Rounding parameter. If set to `None` no rounding is done. Otherwise the scale degrees are rounded to the nearest epsilon
- **sort** – If `True`, sort the output by degree size
- **octave** – The formal octave
- **remove_duplicates** – If `True` remove duplicate entries

The quarter-comma meantone scale is an even-interval scale with the following generating interval:

$$P_5 = \frac{\frac{3}{2}}{\sqrt[4]{\frac{81}{80}}}$$

which is a perfect fifth (in a Pythagorean sense) narrowed by one quarter of the syntonic comma.

An example of use:

```
scale = create_quarter_comma_meantone_scale()
```

yields:

$$\left[1, \quad \frac{8}{25}5^{\frac{3}{4}}, \quad \frac{\sqrt{5}}{2}, \quad \frac{4\sqrt[4]{5}}{5}, \quad \frac{5}{4}, \quad \frac{2}{5}5^{\frac{3}{4}}, \quad \frac{16\sqrt{5}}{25}, \quad \sqrt[4]{5}, \quad \frac{8}{5}, \quad \frac{5^{\frac{3}{4}}}{2}, \quad \frac{4\sqrt{5}}{5}, \quad \frac{5\sqrt[4]{5}}{4}, \quad 2\right]$$

3.2.6 Euler-Fokker Genera

`pytuning.scales.`**`create_euler_fokker_scale`**(*intervals*, *multiplicities*, *octave=2*, *normalize=True*)

Create a scale in the Euler-Fokker Genera

Parameters

- **intervals** – The factors to use for the construction (usually prime numbers)
- **multiplicities** – The multiplicities of the factors (see below)
- **octave** – The formal octave
- **normalize** – If `True`, normalize the intervals to the octave.

`intervals` and `multiplicities` should both be lists of equal length. The entries in `multiplicities` give the number of each factor to use. Therefore the following:

```
intervals      = [3,5,7]
multiplicities = [1,1,1]
scale          = create_euler_fokker_scale(intervals, multiplicities)
```

Will create a scale with one 3, one 5, and one 7 as generators.

The above will produce the following scale:

$$\left[1, \frac{35}{32}, \frac{5}{4}, \frac{21}{16}, \frac{3}{2}, \frac{105}{64}, \frac{7}{4}, \frac{15}{8}, 2\right]$$

Also note that the two statements will generate the same output:

```
intervals      = [3,5,7]
multiplicities = [2,2,1]
scale1         = create_euler_fokker_scale(intervals, multiplicities)

intervals      = [3,3,5,5,7]
multiplicities = [1,1,1,1,1]
scale2         = create_euler_fokker_scale(intervals, multiplicities)

scale1 == scale2
True
```

3.2.7 Diatonic Scales

`pytuning.scales.`**`create_diatonic_scale`**(*generators*, *specification*)

Create a diatonic scale.

Parameters

- **`generators`** – The generator intervals (see below)
- **`specification`** – The scale specification. This is a list of `chars` that correspond to entries in the generators. Note that if all the character representations are a single character, you can pass the specification in as a string for convenience.

Returns The specified scale

`generators` is a list of tuples, the first member of which is an interval specification, the second of which is a character representation. The entries in `specification` should correspond to this value.

As an example, we can create the 12 EDO generators thus:

```
edo12_constructors = [
    (sp.power.Pow(2,sp.Rational(2,12)), "T"),
    (sp.power.Pow(2,sp.Rational(1,12)), "s"),
]
```

We can then create the standard major mode with:

```
create_diatonic_scale(edo12_constructors, ["T","T","s","T","T","T","s"])
```

which will yield:

$$\left[1, \quad \sqrt[6]{2}, \quad \sqrt[3]{2}, \quad 2^{\frac{5}{12}}, \quad 2^{\frac{7}{12}}, \quad 2^{\frac{3}{4}}, \quad 2^{\frac{11}{12}}, \quad 2\right]$$

As another example of creating a diatonic scale, we can use the five-limit constructors (which are defined in `pytuning.constants`):

```
five_limit_constructors = [
    (sp.Rational(16,15), "s"),
    (sp.Rational(10,9),  "t"),
    (sp.Rational(9,8),   "T"),
]
```

to create *Ptolemy's Intense Diatonic Scale*:

```
from pytuning.constants import five_limit_constructors
from pytuning.scales import create_diatonic_scale

scale = create_diatonic_scale(five_limit_constructors,
  ["T", "t", "s", "T", "t", "T", "s"])
```

which gives us:

$$\left[1, \quad \frac{9}{8}, \quad \frac{5}{4}, \quad \frac{4}{3}, \quad \frac{3}{2}, \quad \frac{5}{3}, \quad \frac{15}{8}, \quad 2\right]$$

Note that if every identifier is a single-character string, `specification` can also be passed in as a string. So this is equivalent:

```
from pytuning.constants import five_limit_constructors
from pytuning.scales import create_diatonic_scale

scale = create_diatonic_scale(five_limit_constructors, "TtsTtTs")
```

3.3 Metric Functions

A **metric function** is a function that takes a scale as input and returns a calculated value. As mentioned in *Basic Concepts*, it returns a Python `dict` with the metric name as the key, and the metric value as the value.

The currently defined metrics all estimate the consonance or dissonance of a scale.

3.3.1 sum_p_q()

`pytuning.metrics.`**`sum_p_q`**`(`*scale*`)`

Calculate a metric for a scale

Parameters **`scale`** – The scale.

Returns A `dict` with the metric value.

This is an estimate of scale consonance. It is derived from summing the numerators and denominators of the scale degrees.

Smaller values are more consonant.

Note that this metric looks at the degrees of the scale, so it is somewhat tonic-focused. The similar metric `sum_p_q_for_all_intervals()` is similar, but it sums the numerator and denominator values for all distinct intervals within the scale.

While the metric is numerically defined for ratios expressed as irrational or transcendental numbers, it is really only meaningful for scales with just degrees (ratios expressed as rational numbers).

```
sum_p_q(create_pythagorean_scale())
```

yields:

```
{'sum_p_q': 3138}
```

3.3.2 sum_distinct_intervals()

`pytuning.metrics.`**`sum_distinct_intervals`**(*scale*)

Calculate a metric for a scale

Parameters **`scale`** – The scale.

Returns A `dict` with the metric value.

This metric is an estimate of scale consonance. Numerically it is the number of distinct intervals within the scale (including all ratios and their inversions).

Smaller values are more consonant.

```
sum_distinct_intervals(create_pythagorean_scale())
```

yields:

```
{'sum_distinct_intervals': 22}
```

3.3.3 metric_3()

`pytuning.metrics.`**`metric_3`**(*scale*)

Calculate a metric for a scale

Parameters **`scale`** – The scale.

Returns A `dict` with the metric value.

Metric 3 is an estimate of scale consonance. Given a ratio p/q, it is a heuristic given by the following:

$$m_3 = \sum \frac{1}{\frac{p-q}{q}} = \sum \frac{q}{p-q} \tag{3.1}$$

Smaller values are more consonant.

The summation takes place over all of the intervals in the scale. It does not form a set of distinct intervals.

3.3.4 sum_p_q_for_all_intervals()

`pytuning.metrics.`**`sum_p_q_for_all_intervals`**(*scale*)

Calculate a metric for a scale

Parameters **`scale`** – The scale (i.e., a list of `sympy.Rational` values)

Returns The metric.

This metric is an estimate of scale consonance. It is formed by examining all unique intervals in the scale, and creating a numeric value based upon the summation of the numerators and denominators for all those intervals.

While the metric is numerically defined for ratios expressed as irrational or transcendental numbers, it is really only meaningful for scales with just degrees (ratios expressed as rational numbers).

Smaller values are more consonant.

3.3.5 sum_q_for_all_intervals()

`pytuning.metrics.`**`sum_q_for_all_intervals`**`(`*scale*`)`

Calculate a metric for a scale.

Parameters **`scale`** – The scale (i.e., a list of `Rational` s)

Returns The metric.

Metric 5 is an estimate of scale consonance. It is summation of the denominators of the normalized distinct ratios of the scale.

Smaller values are more consonant.

3.3.6 All Metrics

There is also a function that calculates all defined metrics for a scale.

`pytuning.metrics.`**`all_metrics`**`(`*scale*`)`

Calculate all metrics for the scale

Parameters **`scale`** – The scale (i.e., a list of `Rational` s)

Returns A `dict` containing all metrics.

As an example:

```
pythag = create_pythagorean_scale()
metrics = all_metrics(pythag)
```

will (currently) produce:

```
{
 'metric_3': 49.9049074891784,
 'sum_distinct_intervals': 22,
 'sum_p_q': 3138,
 'sum_p_q_for_all_intervals': 1092732,
 'sum_q_for_all_intervals': 452817
}
```

If new metrics are coded they should be added to the `__all__` data member for inclusion here.

3.4 Scale Analysis and Creation (Redux)

Scale Creation describes the way that many standard scales are generated from within the package. But there are other ways to create scales.

3.4.1 Mode Selection

When one creates a scale – for example, the Pythagorean scale or a scale of the Euler-Fokker Genera – one can looks at the various modes that can be created for that scale and evaluate them by certain criteria.

The `find_best_modes()` function can be used for this. This function accepts and input scale, the number of tones for the mode, and the optimization functions that should be used for evaluating the scale.

As an example, one scale that's I've used in compositions is created from choosing a seven-note mode from a harmonic scale, optimized over the metric `sum_p_q_for_all_intervals()`. This particular scale is based upon the harmonic series referenced to the fourth harmonic.

The following code:

```
harmonic_scale = create_harmonic_scale(4,30)
modes = find_best_modes(harmonic_scale,
                        num_tones=7,
                        sort_order = ['sum_p_q_for_all_intervals'],
                        num_scales=1,
                        metric_function = sum_p_q_for_all_intervals)
```

yields the following object:

```
[{'mask': (0, 2, 4, 5, 8, 12, 14, 15),
  'original_scale': [1,
   17/16,
   9/8,
   19/16,
   5/4,
   21/16,
   11/8,
   23/16,
   3/2,
   25/16,
   13/8,
   27/16,
   7/4,
   29/16,
   15/8,
   2],
  'scale': [1, 9/8, 5/4, 21/16, 3/2, 7/4, 15/8, 2],
  'steps': [2, 2, 1, 3, 4, 2, 1],
  'sum_p_q_for_all_intervals': 572}]
```

The returned scale:

$$\left[1,\quad \frac{9}{8},\quad \frac{5}{4},\quad \frac{21}{16},\quad \frac{3}{2},\quad \frac{7}{4},\quad \frac{15}{8},\quad 2\right]$$

minimizes the metric for all possible combinations of 7 notes chosen from the original harmonic scale.

`pytuning.scale_creation.`**`find_best_modes`**(*scale, num_tones, sort_order=['sum_p_q_for_all_intervals', 'sum_p_q', 'sum_distinct_intervals'], num_scales=1, metric_function=None*)

Find the best modes for a scale, as defined by the specified metrics.

Parameters

- **`scale`** – The scale to analyze
- **`num_tones`** – The number of degrees in the mode
- **`sort_order`** – How the return should be sorted, referenced to the metrics calculated
- **`num_scales`** – The number of scales to return. If `None` all scales will be returned
- **`metric_function`** – The metric function to use. If `None` then `all_metrics` will be used.

Returns A sorted list of mode objects.

The sort order is a list of keys that the metric function should return, applied in order, with an assumption that the lower the metric the more consonant (and "better") the scale. As an example, the default sort order:

```
['sum_p_q_for_all_intervals','sum_p_q','sum_distinct_intervals']
```

Will order the scales by increasing **sum_p_q_for_all_intervals**. If two scales have the same **sum_p_q** value they will be secondarily sorted on **sum_p_q**. If scales have the same **sum_p_q_for_all_intervals** and **sum_p_q** then **sum_distinct_intervals** will be used.

If no metric function is specified the default `all_metrics` will be used. However, for efficiency one may not want to calculate all metrics if they are not being used. For example, if one is just interested in one metric, you can pass the metric directly:

```
from pytuning import create_pythagorean_scale
from pytuning.metrics import sum_p_q_for_all_intervals

pythag = create_pythagorean_scale()
my_metric = lambda scale: dict(sum_p_q(scale), **sum_p_q_for_all_intervals(scale))

best_modes = find_best_modes(pythag, 7, sort_order=['sum_p_q_for_all_intervals'],
                num_scales=1, metric_function=sum_p_q_for_all_intervals)
```

which would yield:

```
[{'mask': (0, 1, 3, 5, 6, 8, 10, 12),
  'original_scale': [1,
   256/243,
   9/8,
   32/27,
   81/64,
   4/3,
   1024/729,
   3/2,
   128/81,
   27/16,
   16/9,
   243/128,
   2],
  'scale': [1, 256/243, 32/27, 4/3, 1024/729, 128/81, 16/9, 2],
  'steps': [1, 2, 2, 1, 2, 2, 2],
  'sum_p_q_for_all_intervals': 4374}]
```

If one is interested in two of the metrics, you could, for example:

```
from pytuning import create_pythagorean_scale
from pytuning.metrics import sum_p_q, sum_p_q_for_all_intervals

pythag = create_pythagorean_scale()
my_metric = lambda scale: dict(sum_p_q(scale), **sum_p_q_for_all_intervals(scale))

best_modes = find_best_modes(pythag, 7, sort_order=['sum_p_q','sum_p_q_for_all_
↪intervals'],
                num_scales=1, metric_function=my_metric)
```

which would yield:

```
[{'mask': (0, 2, 3, 5, 7, 9, 10, 12),
  'original_scale': [1,
   256/243,
   9/8,
   32/27,
   81/64,
   4/3,
   1024/729,
   3/2,
   128/81,
   27/16,
   16/9,
   243/128,
   2],
  'scale': [1, 9/8, 32/27, 4/3, 3/2, 27/16, 16/9, 2],
  'steps': [2, 1, 2, 2, 2, 1, 2],
  'sum_p_q': 161,
  'sum_p_q_for_all_intervals': 4374}]
```

3.4.2 Factoring an Interval

Sometimes it is interesting to take an interval and find an expression for that interval over some set of generator intervals. For this the function `find_factors()` is provided.

One has to specify the generator intervals. This is done by passing the function a list of tuples. Each tuple has two members: The generator interval, and a character representation of the generator interval. Usually these are a single, unique character (such as `X`), but it can also be in the form `1/X`. If it is in this form the generator interval should be the reciprocal of the interval designated by `X`.

As an example, we could create a generator interval that represents the tone and semi-tone of a 31-EDO scale:

```
edo31_constructors = [
    (sp.power.Pow(2,sp.Rational(2,31)), "T"), # a tone
    (sp.power.Pow(2,sp.Rational(1,31)), "s"), # a semitone
]
```

(Note that the tone is just twice the semitone, so we could probably get by with just defining the semitone).

Now we can define an `interval`, say, one of the intervals of the Pythagorean scale:

$$\frac{21}{16}$$

and see what factoring yields an interval closest to the original.

```
results = find_factors(interval, edo31_constructors)
results
```

`results` now contains the factoring, the factoring in symbolic terms, and the resultant interval.

```
([2**(2/31), 2**(2/31), 2**(2/31), 2**(2/31), 2**(2/31), 2**(2/31)],
 ['T', 'T', 'T', 'T', 'T', 'T'],
 2**(12/31))
```

The last entry is the returned interval:

$$2^{\frac{12}{31}}$$

If one is interested in seeing how closely the factored interval matches the original interval, the `ratio_to_cents()` function in `pytuning.utiities` can be used.

```
from pytuning.utilities import ratio_to_cents
print(ratio_to_cents(results[2] / interval))
```

yields:

```
-6.26477830225428
```

In other words, the derived interval is flat about 6.3 cents from the target interval.

`pytuning.utilities.`**`ratio_to_cents`**`(`*ratio*`)`

Convert a scale degree to a cent value

Parameters **`ratio`** – The scale degree (`sympy` value)

Returns The scale degree in cents

Calculates:

$$2^{\left[\frac{1}{1200}\right]}\sqrt{\text{degree}}$$

Note that this function returns a floating point number, not a `sympy` ratio.

3.4.3 Approximating a Scale with Another Scale

The above factoring of an interval over a set of generators can be extended: a scale can be factored too.

To do this the `create_scale_from_scale()` function is used.

The first step in using this function is to create an interval function. It is similar to `find_factors()` in that it accepts an interval and a max factor, and it returns the factor. But the actual generator intervals are bound to this function.

The easiest way of creating this function is to take the generator intervals that you're interested in and to bind them to `find_factors()` via a partial function application. As an example, we can take the five-limit constructors:

```
five_limit_constructors = [
    (sp.Rational(16,15), "s"),
    (sp.Rational(10,9),  "t"),
    (sp.Rational(9,8),   "T"),
]
```

And use them to approximate the Pythagorean scale:

```
from pytuning.scales import create_pythagorean_scale
from pytuning.scale_creation import create_scale_from_scale, find_factors
from pytuning.constants import five_limit_constructors
from functools import partial

interval_function = partial(find_factors, constructors=five_limit_constructors)
pythag = create_pythagorean_scale()
results = create_scale_from_scale(pythag, interval_function)
```

The return value is a tuple, the first element of which is derived scale, the second of which is the symbolic factoring. The scale which was found was

$$\left[1, \quad \frac{16}{15}, \quad \frac{9}{8}, \quad \frac{32}{27}, \quad \frac{81}{64}, \quad \frac{4}{3}, \quad \frac{1024}{729}, \quad \frac{3}{2}, \quad \frac{128}{81}, \quad \frac{27}{16}, \quad \frac{16}{9}, \quad \frac{243}{128}, \quad 2\right]$$

If you look at the Pythagorean scale:

$$\left[1,\quad \frac{256}{243},\quad \frac{9}{8},\quad \frac{32}{27},\quad \frac{81}{64},\quad \frac{4}{3},\quad \frac{1024}{729},\quad \frac{3}{2},\quad \frac{128}{81},\quad \frac{27}{16},\quad \frac{16}{9},\quad \frac{243}{128},\quad 2\right]$$

you can see that they only differ in the second degree (which if we look at the first member of the return we can see is factored as [‘s’]). Looking at how much they differ:

```
ratio = results[0][1] / pythag[1]
print(ratio)
81/80
print(ratio_to_name(ratio))
Syntonic Comma
delta = ratio_to_cents(ratio)
print(delta)
21.5062895967149
```

we see that the difference is $\frac{81}{80}$, which is the syntonic comma (about 21.5 cents).

`create_scale_from_scale()` can also accept a `tone_table` which is a list of the potential breakdowns that can be used in the analysis.

`pytuning.scale_creation.`**`create_scale_from_scale`**(*scale*, *interval_function*, *max_terms=8*, *tone_table=None*)

Given a target scale, calculate the closest matching N-rank linear temperament scale over the provide basis functions.

Parameters

- **`scale`** – The target scale (list or ratios)
- **`interval_function`** – The interval function (see below)
- **`max_terms`** – The maximum number of terms for the factoring along the basis intervals
- **`tone_table`** – A constrained tone table (see below)

Returns A tuple, the first member of which is a list of the derived scale values, and the second of which is a symbolic representation of the factoring found.

The interval function is a function that accepts a degree and `max_terms` and returns the breakdown in the same format as `find_factors`. In general the easiest way to create this function is thought a partial function application of `find_factors` and the specific constructors wanted. As an example, to create a function that will approimate a scale with the five-limit constructors:

```
from pytuning.scale_creation import find_factors, create_scale_from_scale
from pytuning.constants import five_limit_constructors
import functools

find_five_limit_interval = functools.partial(
    find_factors, constructors=five_limit_constructors,
    max_terms=15)

create_five_limit_scale_from_scale = functools.partial(create_scale_from_scale,
                                     interval_function=find_five_limit_interval)
```

if `tone_table` is `None`, the code will perform the factoring with up to `max_terms`. If the tone table is defined only the intervals defined in this table will be used.

The tone table is formatted as a list of tuples, where the members of each tuple are the degree name (`String`), the interval composition (a list of characters taken from the symbolic portion of the constructors), and the value of that factoring. As an example, the tone table that matches published values for the Lucy-tuned scale begins:

```
[('1', [], 1),
 ('5', ['L', 'L', 'L', 's'], sqrt(2)*2**(1/(4*pi))),
 ('2', ['L'], 2**(1/(2*pi))),
 ('6', ['L', 'L', 'L', 'L', 's'], sqrt(2)*2**(3/(4*pi))),
 ('3', ['L', 'L'], 2**(1/pi)),
 ('7', ['L', 'L', 'L', 'L', 'L', 's'], sqrt(2)*2**(5/(4*pi))),
 ('#4', ['L', 'L', 'L'], 2**(3/(2*pi))), ...]
```

(The last member of the tuple is a `sympy` symbolic value.)

With the tone table, the code will return the defined tone which most closely matches the target degree.

Extending the above example, if we were to try to match a Pythagorean scale with an unconstrained factoring of the Five-limit intervals (i.e., with no tone table):

```
pythag = create_pythagorean_scale()
lp = create_five_limit_scale_from_scale(pythag)
```

yields

```
([1,
  16/15,
  9/8,
  32/27,
  81/64,
  4/3,
  1024/729,
  3/2,
  128/81,
  27/16,
  16/9,
  243/128,
  2],
 [[],
  ['s'],
  ['T'],
  ['s', 't'],
  ['T', 'T'],
  ['T', 's', 't'],
  ['s', 's', 't', 't'],
  ['T', 'T', 's', 't'],
  ['T', 's', 's', 't', 't'],
  ['T', 'T', 'T', 's', 't'],
  ['T', 'T', 's', 's', 't', 't'],
  ['T', 'T', 'T', 'T', 's', 't'],
  ['T', 'T', 'T', 's', 's', 't', 't']])
```

Note that the first entry of the factors is always for the ratio 1, and is returned as an empty list (as there really *are* no factors in this sense).

3.5 Tuning Tables

A **tuning table** is a text representation of a scale that can be interpreted by an external program. This is usually used to tune a synthesizer or other sound source so that the scale can be used in a musical composition.

Some tuning tables – such as that used by Scala – describe the scale in absolute terms, but most need to have a reference defined so that the scale degrees can be mapped to a frequency. For this purpose the PyTuning package has adopted the MIDI standard for note numbers and frequencies.

Octave	C	C#	D	D#	E	F	F#	G	G#	A	A#	B
-2	0	1	2	3	4	5	6	7	8	9	10	11
-1	12	13	14	15	16	17	18	19	20	21	22	23
0	24	25	26	27	28	29	30	31	32	33	34	35
1	36	37	38	39	40	41	42	43	44	45	46	47
2	48	49	50	51	52	53	54	55	56	57	58	59
3	60	61	62	63	64	65	66	67	68	69	70	71
4	72	73	74	75	76	77	78	79	80	81	82	83
5	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107
7	108	109	110	111	112	113	114	115	116	117	118	119
8	120	121	122	123	124	125	126	127				

So, for example, note number 69 corresponds to middle A, whereas 60 corresponds to middle C. The frequency standard is 12-EDO, and the note 69 is pegged to 440 Hz. Thus if you passed `69` as the reference note, the 69'th entry in the table would be 440 Hz and this would correspond to the first degree of the scale. `60` would cause the first degree of the scale to be assigned to that note number (with a corresponding frequency of about 261.6 Hz). The timidity soft synth is an example of a synthesizer that needs this reference note and frequency.

In general the table will need to be output to disk so that it can read by the program. This can be done with something like:

```
from pytuning.tuning_tables import create_timidity_tuning
from pytuning.scales import create_euler_fokker_scale

reference_note = 60
scale = create_euler_fokker_scale([3,5],[3,1])

tuning = create_timidity_tuning(scale, reference_note=reference_note)

with open("timidity.table", "w") as table:
    table.write(tuning)
```

This will cause the generated scale:

$$\left[1, \quad \frac{135}{128}, \quad \frac{9}{8}, \quad \frac{5}{4}, \quad \frac{45}{32}, \quad \frac{3}{2}, \quad \frac{27}{16}, \quad \frac{15}{8}, \quad 2\right]$$

to be written to a disk file, `timidity.table`, which can be understood by timidity:

```
timidity -Z timidity.table score.mid
```

3.5.1 Timidity

`pytuning.tuning_tables.`**`create_timidity_tuning`**(*scale*, *reference_note=60*, *reference_frequency=None*)

Create a Timidity++ tuning table

Parameters

- **`scale`** – The scale to model (list of frequency ratios)
- **`reference_note`** – The MIDI number of the absolute frequency reference

- **reference_frequency** – The frequency of the reference note. If None (the default) the frequency will be taken from the standard MIDI 12-EDO tuning

Returns A Timidity tuning table as a String

The default value of reference_note pegs the scale to to the standard concert tuning of middle C (A = 440Hz).

The Timidity table is basically a list of integers for all defined MIDI note numbers, with each entry as 1000 times the note frequency

As a somewhat detailed example, let's say that the user had a 12-EDO scale constructed, and wanted to pin the tonic note to the standard A440. The following will do this:

```
from pytuning.scales import create_edo_scale
from pytuning.tuning_tables import create_timidity_tuning

scale = create_edo_scale(12)
tuning_table = create_timidity_tuning(scale, reference_note=69)
```

with the first part of the table given by:

```
# Timidity tuning table created by pytuning,
# call timidity with the -Z option to enable.
# Note reference: 69; Freq reference: 440.000000 Hz
8176
8662
9177
9723
10301
10913
11562
12250
12978
```

To use the table, one starts Timidity with the **-Z** option, i.e:

```
timidity -Z table.name -iA
```

3.5.2 Scala

The Scala tuning table can be used with the Scala package, but it can also be used to tune the soft synth Zynaddsubfx, as well as its derivative Youshimi. With the soft synths you will need to explicitly set the reference note and frequency in the scale GUI.

pytuning.tuning_tables.**create_scala_tuning**(*scale*, *name*)
Create a Scala scale file

Parameters

- **scale** – The scale (list of frequency ratios)
- **name** – The name of the scale

Returns A Scala file as a String

The Scala file can be used to tune various things, most germane being Yoshimi. However, keep in mind that the Scala file does **not** include a base note or frequency, so for tuning purposes those data will need to be captured or input in some other way.

As an example of use, the Scala file for the default Pythagorean tuning can be calculated thus:

```python
from pytuning.scales.pythagorean import create_pythagorean_scale
from pytuning.tuning_tables import create_scala_tuning

scale = create_pythagorean_scale()
table = create_scala_tuning(scale,"Pythagorean Tuning")
```

which yields:

```
! Scale produced by pytuning. For tuning yoshimi or zynaddsubfx,
! only include the portion below the final '!'
!
Pythagorean Tuning
12
!
256/243
9/8
32/27
81/64
4/3
1024/729
3/2
128/81
27/16
16/9
243/128
2/1
```

Note that the Scala file uses exact ratios where possible, otherwise it will convert to a cent value. Thus the code:

```python
from pytuning.scales import create_edo_scale
from pytuning.tuning_tables import create_scala_tuning

scale = create_edo_scale(12)
table = create_scala_tuning(scale,"12-TET Tuning")
```

will produce:

```
! Scale produced by pytuning. For tuning yoshimi or zynaddsubfx,
! only include the portion below the final '!'
12-TET Tuning
12
!
100.00000
200.00000
300.00000
400.00000
500.00000
600.00000
700.00000
800.00000
900.00000
1000.00000
1100.00000
2/1
```

3.5.3 Fluidsynth

`pytuning.tuning_tables.`**`create_fluidsynth_tuning`**(*scale, reference_note=60, chan=[0], bank=0, prog=[0], reference_frequency=None*)

Create a Fluidsynth tuning table

Parameters

- **`scale`** – The scale to model (list of frequency ratios)
- **`reference_note`** – The MIDI number of the absolute frequency reference
- **`reference_frequency`** – The frequency of the reference note. If `None` (the default) the frequency will be taken from the standard MIDI 12-EDO tuning
- **`chan`** – A list of channels for which to create the table
- **`bank`** – The bank for the tuning table
- **`prog`** – A list of program numbers for the tuning table

Returns A Fluidsynth tuning table as a `String`

The default value of `reference_note` pegs the scale to to the standard concert tuning of middle C (A = 440Hz).

The Fluidsyny tuning model allows each channel, bank, and program to have a different tuning. Thus, if one, say, wants all programs to be tuned to the scale, the tuning table can get quite large.

As a somewhat detailed example, let's say that the user had a 12-EDO scale constructed, and wanted to pin the tonic note to the standard A440. The following will do this:

```
from pytuning.scales import create_edo_scale
from pytuning.tuning_tables import create_timidity_tuning

scale = create_edo_scale(12)
tuning_table = create_fluidsynth_tuning(scale, prog=range(128), reference_note=69)
```

with the first part of the table given by:

```
# Fluidsynth Tuning Table created by pytuning
# Note reference: 69; Freq reference: 440.000000 Hz
tuning tuning000 0 0
tune 0 0 0 0.000000
tune 0 0 1 100.000000
tune 0 0 2 200.000000
tune 0 0 3 300.000000
tune 0 0 4 400.000000
tune 0 0 5 500.000000
tune 0 0 6 600.000000
tune 0 0 7 700.000000
tune 0 0 8 800.000000
tune 0 0 9 900.000000
```

To use the table, one should start fluidsynth with the **-f** option:

```
fluidsynth -f table.name
```

3.5.4 Csound

For use in Csound PyTuning will generate a table of frequencies that can be used as a table lookup, mapped to MIDI note number. As mentioned in the basic concepts, the easiest way to use this is via the `table` opcode:

```
inote     init          p4
iveloc    init          p5
ifreq     table         inote, 1
a1        oscil         iveloc, ifreq, 2
          outs          a1, a1
```

`pytuning.tuning_tables.`**`create_csound_tuning`**(*scale*, *reference_note=60*, *reference_frequency=None*, *table_num=1*)

Create a CSound tuning table

Parameters

- **`scale`** – The scale (list of frequency ratios)
- **`reference_note`** – The MIDI number of the absolute frequency reference
- **`reference_frequency`** – The frequency of the reference note. If `None` (the default) the frequency will be taken from the standard MIDI 12-EDO tuning
- **`table_num`** – The **f** table number to use

CSound has many ways of generating microtonalities. For `pytuning` a table lookup keyed on MIDI note number is used.

As an example of use, let's say that we want to use the 12-EDO scale with the tonic at A440:

```
from pytuning.scales import create_edo_scale
from pytuning.tuning_tables import create_timidity_tuning

scale = create_edo_scale(12)
table = create_csound_tuning(scale, reference_note=69)
```

This will produce the following output, which can be included in the CSound score file:

```
f1 0 256 -2     8.17580     8.66196     9.17702     9.72272    10.30086    10.
↪91338    11.56233    12.24986 \
                12.97827    13.75000    14.56762    15.43385    16.35160    17.
↪32391    18.35405    19.44544 \
                20.60172    21.82676    23.12465    24.49971    25.95654    27.
↪50000    29.13524    30.86771 \
                32.70320    34.64783    36.70810    38.89087    41.20344    43.
↪65353    46.24930    48.99943 \
                51.91309    55.00000    58.27047    61.73541    65.40639    69.
↪29566    73.41619    77.78175 \
                82.40689    87.30706    92.49861    97.99886   103.82617   110.
↪00000   116.54094   123.47083 \
               130.81278   138.59132   146.83238   155.56349   164.81378   174.
↪61412   184.99721   195.99772 \
               207.65235   220.00000   233.08188   246.94165   261.62557   277.
↪18263   293.66477   311.12698 \
               329.62756   349.22823   369.99442   391.99544   415.30470   440.
↪00000   466.16376   493.88330 \
               523.25113   554.36526   587.32954   622.25397   659.25511   698.
↪45646   739.98885   783.99087 \
               830.60940   880.00000   932.32752   987.76660  1046.50226  1108.
↪73052  1174.65907  1244.50793 \
```

(continues on next page)

(continued from previous page)

```
            1318.51023  1396.91293  1479.97769  1567.98174  1661.21879  1760.
↪00000  1864.65505  1975.53321 \
            2093.00452  2217.46105  2349.31814  2489.01587  2637.02046  2793.
↪82585  2959.95538  3135.96349 \
            3322.43758  3520.00000  3729.31009  3951.06641  4186.00904  4434.
↪92210  4698.63629  4978.03174 \
            5274.04091  5587.65170  5919.91076  6271.92698  6644.87516  7040.
↪00000  7458.62018  7902.13282 \
            8372.01809  8869.84419  9397.27257  9956.06348 10548.08182 11175.
↪30341 11839.82153 12543.85395
```

3.6 Number Theory

While this part of the package isn't particularly fleshed out yet, there are a few number-theoretic functions for the analysis of scales.

3.6.1 Odd Limits

PyTuning contains functions for finding the odd Limit for both intervals and scales.

We can define and interval – say, $\frac{45}{32}$, and find its odd-limit with the following:

```
from pytuning.number_theory import odd_limit

interval = sp.Rational(45,32)
limit = odd_limit(interval)
```

which yields and answer of 45.

One can also find the odd limit of an entire scale with the `find_odd_limit_for_scale()` function:

```
from pytuning.scales import create_euler_fokker_scale
from pytuning.number_theory import find_odd_limit_for_scale

scale = create_euler_fokker_scale([3,5],[3,1])
limit = find_odd_limit_for_scale(scale)
```

which yields 135. (Examining the scale:

$$\left[1, \quad \frac{135}{128}, \quad \frac{9}{8}, \quad \frac{5}{4}, \quad \frac{45}{32}, \quad \frac{3}{2}, \quad \frac{27}{16}, \quad \frac{15}{8}, \quad 2\right]$$

you will see that this is the largest odd number, and is found in the second degree.)

`pytuning.number_theory.`**`odd_limit`**`(i)`
Find the odd-limit of an interval.

Parameters **`i`** – The interval (a `sympy.Rational`)

Returns The odd-limit for the interval.

`pytuning.number_theory.`**`find_odd_limit_for_scale`**`(s)`
Find the odd-limit of an interval.

Parameters **`s`** – The scale (a list of `sympy.Rational` values)

Returns The odd-limit for the scale.

3.6.2 Prime Limits

One can also compute prime limits for both scales and intervals. Extending the above example, one would assume that the Euler-Fokker scale would have a prime-limit of 5, since that's the highest prime used in the generation, and in fact:

```
from pytuning.scales import create_euler_fokker_scale
from pytuning.number_theory import find_prime_limit_for_scale

scale = create_euler_fokker_scale([3,5],[3,1])
limit = find_prime_limit_for_scale(scale)
```

will return 5 as the limit.

`pytuning.number_theory.`**`prime_limit`**(*i*)

Find the prime-limit of an interval.

Parameters **`i`** – The interval (a `sympy.Rational`)

Returns The prime-limit for the interval.

`pytuning.number_theory.`**`find_prime_limit_for_scale`**(*s*)

Find the prime-limit of an interval.

Parameters **`s`** – The scale (a list of `sympy.Rational` values)

Returns The prime-limit for the scale.

3.6.3 Prime Factorization of Degrees

PyTuning has functions for breaking a ratio down into prime factors, and the inverse of reassembling them.

`pytuning.number_theory.`**`prime_factor_ratio`**(*r*, *return_as_vector=False*)

Decompose a ratio (degree) to prime factors

Parameters

- **`r`** – The degree to factor (`sympy.Rational`)
- **`return_as_vector`** – If `True`, return the factors as a vector (see below)

Returns The factoring of the ratio

By default this function will return a `dict`, with each key being a prime, and each value being the exponent of the factorization.

As an example, the syntonic comma, $\frac{81}{80}$, can be factored:

```
r = syntonic_comma
q = prime_factor_ratio(r)
print(q)
{2: -4, 3: 4, 5: -1}
```

which can be interpreted as:

$$\frac{3^4}{2^4 \cdot 5^1}$$

If `return_as_vector` is `True`, the function will return a tuple, with each position a successive prime. The above example yields:

```
r = syntonic_comma
q = prime_factor_ratio(r, return_as_vector=True)
print(q)
(-4, 4, -1)
```

Note that zeros will be included in the vector, thus:

```
r = sp.Rational(243,224)
q = prime_factor_ratio(r, return_as_vector=True)
print(q)
(-5, 5, 0, -1)
q = prime_factor_ratio(r)
print(q)
{2: -5, 3: 5, 7: -1}
```

pytuning.number_theory.**create_ratio_from_primes**(*factors*)

Given a prime factorization of a ratio, reconstruct the ratio. This function in the inverse of *prime_factor_ratio()*.

Parameters **factors** – The factors. Can be a dict or a tuple (see prime_factor_ratio for a discussion).

Returns The ratio (sympy value).

3.7 Utilities

The PyTuning package contains some utilities which may be useful. In general these tend to be smaller utilities and tasks that are useful in the analysis of musical scales, but they are not full-featured "things" in and of themselves.

3.7.1 Interval Normalization

pytuning.utilities.**normalize_interval**(*interval*, *octave=2*)

Normalize a musical interval

Parameters

- **interval** – The interval to normalize. Should be a frequency ratio, most usefully expressed as a *sympy.Rational* or related data item
- **octave** – The formal octave. Defaults to 2

Returns The interval, normalized

Note that any formal octave can be used. In normal usage a 2 will be used (i.e., a doubling of frequency is an octave).

Normalization works by finding the smallest power of two (or octave) that when multiplied by the interval (in the case of an interval less than 1) or divided into the interval (for intervals greater than 2) will bring the interval into the target range of $1 \leq i \leq 2$.

As an example, the interval 9 would normalize to $\frac{9}{8}$, because 9 needs to be scaled down by three octaves to fall within the limit of 1 and 2:

$$\frac{9}{8} = \frac{9}{2^3}$$

```
ni = normalize_interval(sp.Integer(9))
print(ni)
9/8
```

One can also normalize on a non-standard interval, for example, 3:

```
ni = normalize_interval(sp.Integer(34), octave=3)
print(ni)
34/27
```

3.7.2 Distinct Intervals

`pytuning.utilities.`**`distinct_intervals`**`(`*scale*`)`

Find the distinct intervals in a scale, including inversions

Parameters **`scale`** – The scale to analyze

Returns A list of distinct intervals

The scale should be specified as a list of `sympy` numerical values (`Rational` or `Integer`). Note that the convention adopted in this code is that scale[0] is a unison and scale[-1] is the formal octave (often 2).

As an example of a valid scale, a standardized Pythagorean tuning could be passed into the function:

$$\left[1, \frac{256}{243}, \frac{9}{8}, \frac{32}{27}, \frac{81}{64}, \frac{4}{3}, \frac{1024}{729}, \frac{3}{2}, \frac{128}{81}, \frac{27}{16}, \frac{16}{9}, \frac{243}{128}, 2\right]$$

If one were hand-crafting this scale, it would look something like:

```
import sympy as sp
scale = [sp.Integer(1), sp.Rational(256,243), sp.Rational(9,8), ...]
```

The function returns a list in rational/symbolic terms. If numerical values are needed, one can, for example, map `ratio_to_cents` to obtain it:

```
di = distinct_intervals(scale)
di_in_cents = [ratio_to_cents(x) for x in di]
```

`distinct_intervals()` returns all the distinct intervals within a musical scale. Note, though, that it does not include the unison (or the octave) in the results, as all scales contain those intervals by definitions.

As an example, if we were to take a Pythagorean scale and find the intervals that exist within it:

```
pythag = create_pythagorean_scale()
di = distinct_intervals(pythag)
```

we end up with:

$$\left[\frac{2187}{2048}, \frac{256}{243}, \frac{8192}{6561}, \frac{262144}{177147}, \frac{4096}{2187}, \frac{3}{2}, \frac{243}{128}, \frac{1024}{729}, \frac{19683}{16384}, \frac{729}{512}, \frac{6561}{4096}, \frac{65536}{59049}, \frac{177147}{131072}, \frac{59049}{32768}, \frac{81}{64}, \right.$$

3.7.3 Converting a Ratio to a Cent Value

`pytuning.utilities.`**`ratio_to_cents`**`(`*ratio*`)`

Convert a scale degree to a cent value

Parameters **`ratio`** – The scale degree (`sympy` value)

Returns The scale degree in cents

Calculates:

$$\sqrt[2\left[\frac{1}{1200}\right]]{\text{degree}}$$

Note that this function returns a floating point number, not a `sympy` ratio.

This function is useful if you have a symbolic value (a rational or transcendental, for example) and you want to see its value in cents (a logarithmic scale in which there are 1200 steps in a factor of two). For example:

```
interval = sp.Rational(3,2) # A perfect fifth
cents = ratio_to_cents(interval)
print(cents)
701.955000865387
```

3.7.4 Converting a Cent Value to a Ratio

`pytuning.utilities.`**`cents_to_ratio`**`(`*cents*`)`

Convert a cent value to a ratio

Parameters **`cents`** – The degree value in cents

Returns the frequency ratio

This function takes a cent value and returns it as a frequency ratio (a `sympy` floating point number).

```
print(cents_to_ratio(700.0))
1.49830707687668
```

(In other words, the 12-EDO fifth (700 cents) is very close to that of the Pythagorean fifth ($\frac{3}{2}$, or 1.5).)

3.7.5 Converting a Note Number to a Frequency

`pytuning.utilities.`**`note_number_to_freq`**`(`*note*, *scale=None*, *reference_note=69*, *reference_frequency=440.0*`)`

Convert a note number (MIDI) to a frequency (Hz).

Parameters

- **`note`** (*`reference`*) – The note number (0<=note<=127)
- **`scale`** – The scale. If none it assume EDO 12.
- **`note`** – The conversions reference note
- **`reference_frequency`** – The frequency of the reference note

Returns The frequency of the note in Hertz

The default values for `reference_note` and `reference_frequency` correspond to standard orchestral tuning, a4 = 440 Hz.

With this function we can calculate the frequency of any note number. If defaults to the MIDI standard, which pegs note number 69 to 440 Hz and uses a 12-EDO scale.

As an example, MIDI note 60 (Middle-C):

```
print(note_number_to_freq(60))
261.625565300599
```

But if, for example, we wanted to use a different pitch standard, we could peg A to 444 Hz.

```
print(note_number_to_freq(60, reference_frequency=444.0))
264.003979530604
```

You can also pass in a non-EDO tuning if you're converting a different kind of scale to frequencies. This is used often in the code associated with the tuning tables.

3.7.6 Naming A Ratio

`pytuning.utilities.`**`ratio_to_name`**`(`*ratio*`)`

Convert a scale degree to a name

Parameters **`ratio`** – The input scale degree (a `sympy` value)

Returns The degree name if found, `None` otherwise

This function will look up the name of a ratio and return it (returning `None`) if it is not found.

As an example:

```
pythag = create_pythagorean_scale()
names = [ratio_to_name(x) for x in pythag]
```

`names` now contains:

```
 ['Unison',
'Pythagorean Minor Second',
'Pythagorean Major Second',
'Pythagorean Minor Third',
'Pythagorean Major Third',
'Perfect Fourth',
'Pythagorean Diminished Fifth',
'Perfect Fifth',
'Pythagorean Minor Sixth',
'Pythagorean Major Sixth',
'Pythagorean Minor Seventh',
'Pythagorean Major Seventh',
'Octave']
```

There are currently about 260 intervals in the internal catalog, so while not complete, the database is fairly extensive.

3.7.7 Comparing Two Scales

`pytuning.utilities.`**`compare_two_scales`**`(`*scale1, scale2, reference_freq=220.0, title=['Scale1', 'Scale2']*`)`

Compare two scales

param scale1 The first scale (list of `sympy` values)

param scale2 The second scale (list of `sympy` values)

param reference_freq The frequency (Hz) of the first degree

param title The scale names (list of strings with len = 2)

returns `None`, (ie nothing)

This function will produce a simple textual representation of the difference between two scales. As an example, comparing the 12-EDO and Pythagorean scales:

```
from pytuning.scales import create_edo_scale, create_pythagorean_scale
from pytuning.utilities import compare_two_scales

scale_1 = create_edo_scale(12)
scale_2 = create_pythagorean_scale()

compare_two_scales(scale_1, scale_2, title=['12-TET', 'Pythagorean'])
```

produces:

```
          12-TET               Pythagorean
    Cents        Freq       Cents        Freq  Delta(Cents)
=========   =========   =========   =========  ============
   0.0000    220.0000      0.0000    220.0000        0.0000
 100.0000    233.0819     90.2250    231.7695        9.7750
 200.0000    246.9417    203.9100    247.5000       -3.9100
 300.0000    261.6256    294.1350    260.7407        5.8650
 400.0000    277.1826    407.8200    278.4375       -7.8200
 500.0000    293.6648    498.0450    293.3333        1.9550
 600.0000    311.1270    588.2700    309.0261       11.7300
 700.0000    329.6276    701.9550    330.0000       -1.9550
 800.0000    349.2282    792.1800    347.6543        7.8200
 900.0000    369.9944    905.8650    371.2500       -5.8650
1000.0000    391.9954    996.0900    391.1111        3.9100
1100.0000    415.3047   1109.7750    417.6562       -9.7750
1200.0000    440.0000   1200.0000    440.0000        0.0000
```

3.8 Visualizations

PyTuning includes some visualizations for the analysis of scales, but this is not yet fleshed out.

The graphics are via Matplotlib, which is the *de facto* implementation of analytical graphics for python. There is also an optional dependency on Seaborn. Seaborn is good to include if you can – it makes the graphs better – but it's a large package, owning to its dependencies (which include SciPy and Pandas; great packages, but extensive and large). If you have disk space to spare you may want to install it; otherwise you can get my with Matplotlib alone. On my system SciPy and Pandas weigh in at about 200 megabytes, not including any dependencies that *they* require. Matplotlib, on the other hand, is about 26 megabytes.

3.8.1 The Consonance Matrix

It's nice to be able to get an intuitive feeling for the consonance or dissonance of a scale. For this, the **consonance matrix** is provided.

The consonance matrix forms an interval between every degree in the scale and applies a metric function to it. The default metric function just measures the denominator of the interval after any simplification it can undergo (and is really only meaningful for degrees which are expressed an integer ratios).

As an example, we can create a scale of the Euler-Fokker type:

```
scale = create_euler_fokker_scale([3,5],[2,1])
```

which is

$$\left[1, \quad \frac{9}{8}, \quad \frac{5}{4}, \quad \frac{45}{32}, \quad \frac{3}{2}, \quad \frac{15}{8}, \quad 2\right]$$

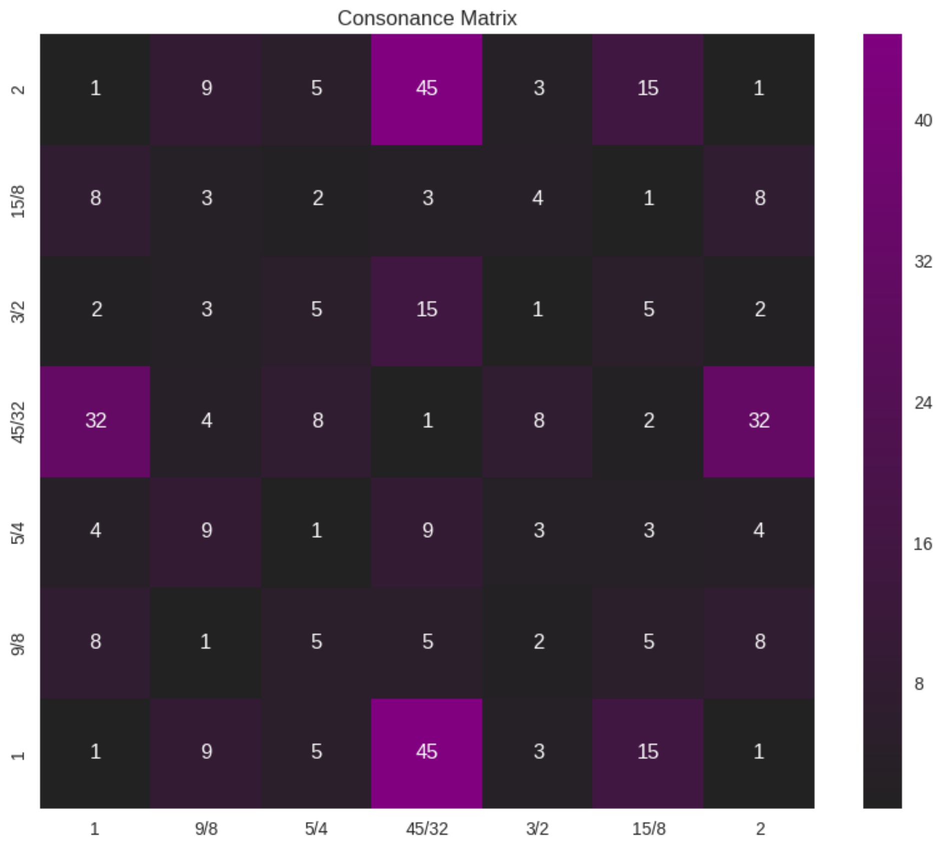
Now, to create a consonance matrix:

```
from pytuning.visualizations import consonance_matrix
matrix = consonance_matrix(scale)
```

`matrix` now contains a matplotlib `Figure`. To write it as PNG:

```
matrix.savefig('consonance.png')
```

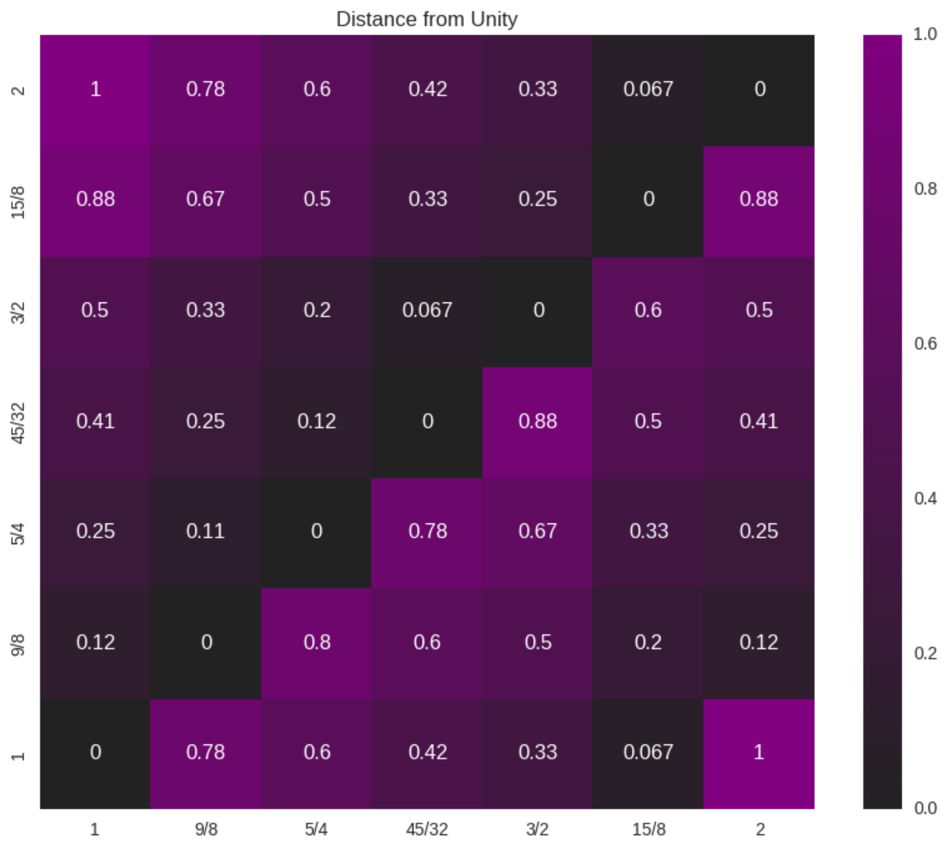
Which yields:



Because this function accepts an arbitrary metric, it can be used for any analysis performed on the intervals of a scale; it does not need to really be a measurement or estimate of consonance. As an example, let's say that (for some reason) you're interested in how far from unity each interval is:

```
def metric_unity(degree):
    normalized_degree = normalize_interval(degree)
    y = abs (1.0 - normalized_degree.evalf())
    return y

scale = create_euler_fokker_scale([3,5],[2,1])
matrix = consonance_matrix(scale, metric_function=metric_unity, title="Distance from␣
↪Unity")
```

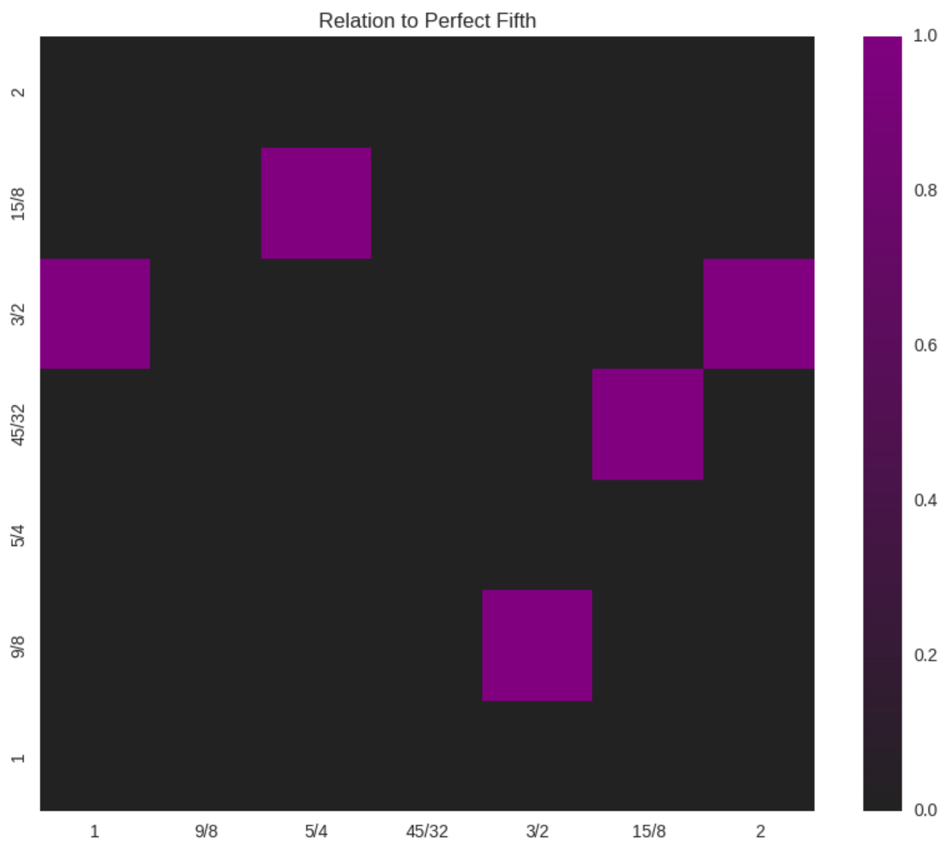
Our graph now looks like:



We could even do something like look for perfect fifths within our scale:

```
def metric_fifth(degree):
    p5 = sp.Rational(3,2)
    normalized_degree = normalize_interval(degree)
    y = normalize_interval(p5 / normalized_degree).evalf()
    y = y if y == 1 else 0
    return y

scale = create_euler_fokker_scale([3,5],[2,1])
matrix = consonance_matrix(scale, metric_function=metric_fifth,
  title="Relation to Perfect Fifth", annot=False)
```

which gives us:



where the highlighted cells denote perfect fifths.

You'll note that the entry for $\frac{45}{32}$ and $\frac{15}{8}$ is 1 (a perfect fifth). We can verify this:

```
print(normalize_interval(sp.Rational(45,32)  / sp.Rational(15,8)))
3/2
```

pytuning.visualizations.**consonance_matrix**(*scale*, *metric_function=None*, *figsize=(10, 8)*, *title='Consonance Matrix'*, *annot=True*, *cmap=None*, *fig=None*, *vmin=None*, *vmax=None*)

Display a consonance matrix for a scale

Parameters

- **scale** – The scale to analyze (list of frequency ratios)
- **metric_function** – The metric function (see below)
- **figsize** – Size of the figure (tuple, in inches)
- **title** – the graph title
- **annot** – If *True*, display the value of the metric in the grid cell
- **cmap** – A custom `matplotlib` colormap, if desired

- **fig** – a `matplotlib.figure.Figure` or `None`. If `None` a figure will be created
- **vmin** – If secified, the lowest value of the range to plot. Only works if Seaborn is included.
- **vman** – If secified, the largest value of the range to plot. Only works if Seaborn is included.

Returns a `matplotlib.Figure` for display

The consonance matrix is created by taking each scale degree along the bottom and left edges of a matrix, forming a frequency ratio between the left and bottom value, and applying the metric function to that value.

The default metric function is the denominator of the normalized ratio – the thought being that the smaller this number is the more consonant the interval.

If a user-specified metric function is used, that function should accept one parameter (the ratio) and return one value. As an example, the default metric function:

```
def metric_denom(degree):
    normalized_degree = normalize_interval(degree)
    return sympy.fraction(normalized_degree)[1]
```

Note that, as with most parts of this code, the ratios should be expressed in terms of `sympy` values, usually `sympy.Rational`'s

A note of figures: With the default of fig=None, this function will just produce a figure and return it for display. However, if one wants to plot multiple figures and can pre-allocate it. This is useful, for example, if one wants to plot multiple figures on the same chart. As an example, the following will plot two separate consonance matrices, side-by-side:

```
import matplotlib.pyplot as plt
fig = plt.figure(figsize=(15,6))
plt.subplot(plt.GridSpec(1, 2)[0,0])
fig = consonance_matrix(scale,title="Full Scale", fig=fig)
plt.subplot(plt.GridSpec(1, 2)[0,1])
fig = consonance_matrix(mode_scale,title="Mode Scale", fig=fig)
fig
```

`vmin` and `vmax` are useful if you're plotting multiple graphs in the same figure; they can be used to ensure that all the component graphs use the same scale, so that visually the graphs are related. Otherwise each graph will have its own scale.

3.9 Using PyTuning Interactively

With a little knowledge of Python, one can use PyTuning in an interactive environment.

If you plan on doing this, I recommend using the Jupyter QtConsole. Jupyter is a full-featured interactive environment for several programming languages (the project started as IPython, but has expanded to cover many languages).

Included in the distribution is a script, `interactive.py`, which is useful for setting up your environment.

3.9.1 Installing Jupyter

(Note: I have experience with Linux and MacOS, so these instructions are focused on these platforms. Jupyter runs under Windows, but I have no experience on that platform.)

On Linux, good ways of installing Jupyter include using your native package manager or installing it via a third-party distribution.

Native Packages

On Ubuntu Jupyter is still referred to as IPython. On Xenial, for example, there are packages for both Python 2.7 and Python 3:

```
vagrant@ubuntu-xenial:~$ aptitude search qtconsole
p   ipython-qtconsole                       - enhanced interactive Python shell - Qt␣
 ↪console
p   ipython3-qtconsole                      - enhanced interactive Python 3 shell -␣
 ↪Qt console
vagrant@ubuntu-xenial:~$
```

On Arch Linux:

```
mark@lucid:~$ pacman -Ss qtconsole
community/python-qtconsole 4.2.1-1
    Qt-based console for Jupyter with support for rich media output
community/python2-qtconsole 4.2.1-1
    Qt-based console for Jupyter with support for rich media output
mark@lucid:~$
```

(PyTuning will run under either Python 2.7 or Python 3.X, so the version you install is up to you.)

I would also suggest installing Matplotlib so that graphics can be used within the console, i.e:

```
vagrant@ubuntu-xenial:~$ aptitude search matplotlib
p   python-matplotlib                        - Python based plotting system in a␣
 ↪style similar to
p   python-matplotlib:i386                   - Python based plotting system in a␣
 ↪style similar to
p   python-matplotlib-data                   - Python based plotting system (data␣
 ↪package)
p   python-matplotlib-dbg                    - Python based plotting system (debug␣
 ↪extension)
p   python-matplotlib-dbg:i386               - Python based plotting system (debug␣
 ↪extension)
p   python-matplotlib-doc                    - Python based plotting system␣
 ↪(documentation packag
p   python-matplotlib-venn                   - Python plotting area-proportional two-␣
 ↪and three-w
p   python3-matplotlib                       - Python based plotting system in a␣
 ↪style similar to
p   python3-matplotlib:i386                  - Python based plotting system in a␣
 ↪style similar to
p   python3-matplotlib-dbg                   - Python based plotting system (debug␣
 ↪extension, Pyt
p   python3-matplotlib-dbg:i386              - Python based plotting system (debug␣
 ↪extension, Pyt
p   python3-matplotlib-venn                  - Python 3 plotting area-proportional␣
 ↪two- and three
vagrant@ubuntu-xenial:~$
```

And optionally Seaborn:

```
vagrant@ubuntu-xenial:~$ aptitude search seaborn
p   python-seaborn                           - statistical visualization library
p   python3-seaborn                          - statistical visualization library
vagrant@ubuntu-xenial:~$
```

(But note that Seaborn is a bit large. See the discussion in *Visualizations*.)

Third Party Packages

Jupyter can also be installed via third-party Python distributions. This is my preferred way of doing it, and on MacOS it is (in my opinion) the only viable option. I imagine that a third-party distribution would be the easiest way to do this on Windows.

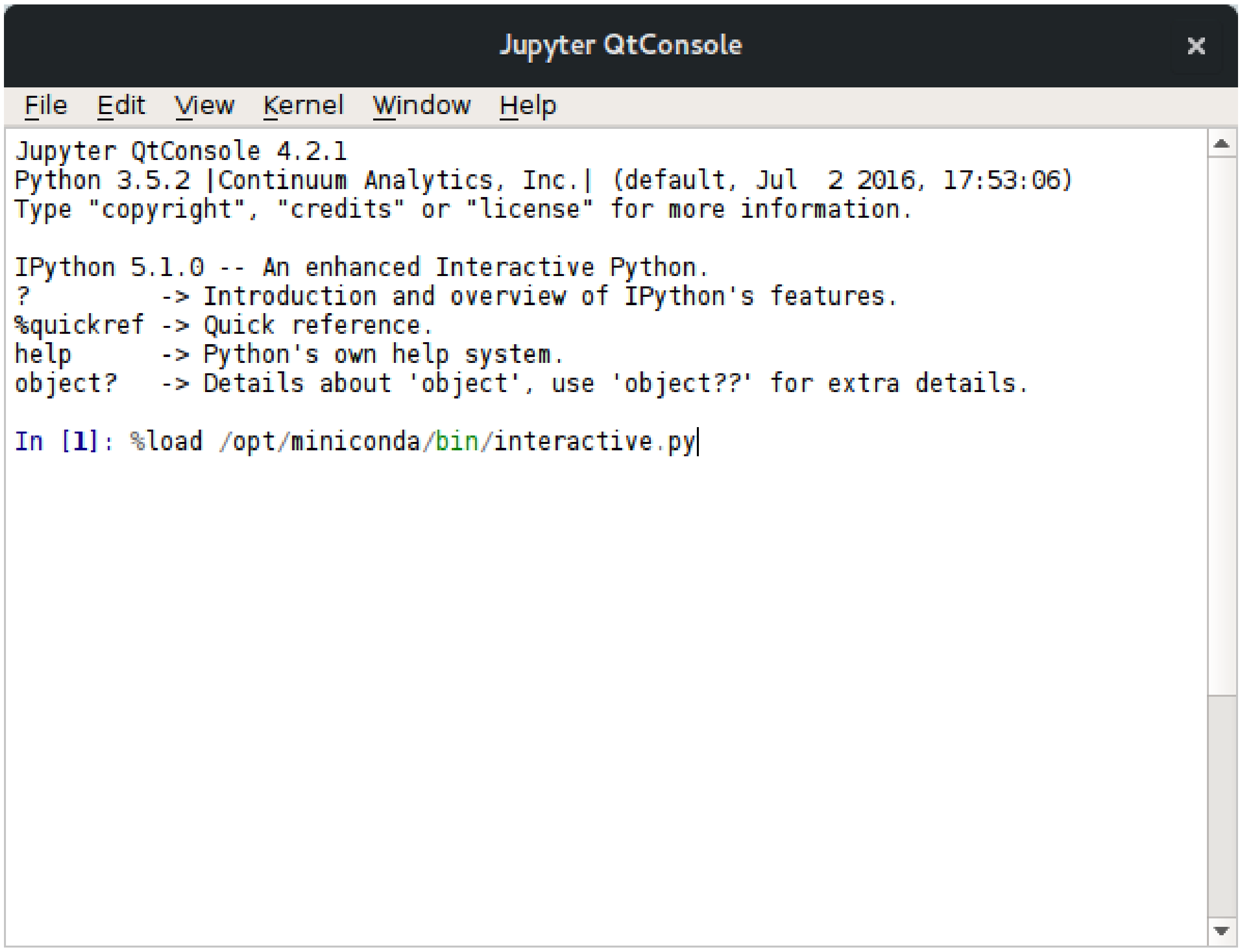
One good distribution is Continuum Analytics Miniconda. Once miniconda is installed, the `conda` tool can be used to install the necessary packages:

```
vagrant@ubuntu-xenial:~$ conda install jupyter qtconsole matplotlib seaborn sympy␣
↪numpy
```

3.9.2 Setting the Environment

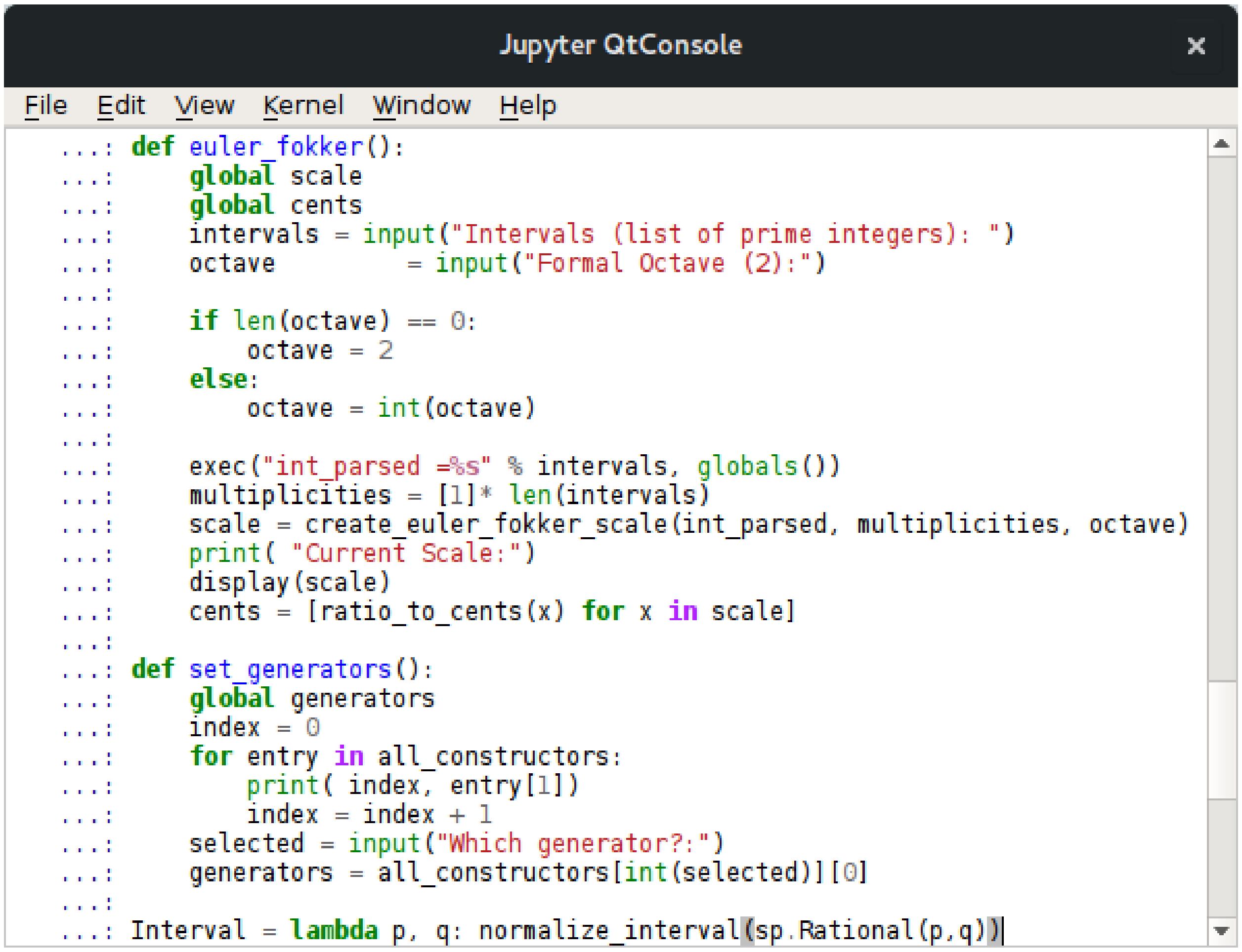
The PyTuning distribution contains a script, `interactive.py`, that can be used to import the package into your namespace, as well as setting up some convenience functions. Where that script lives on your computer can vary by platform as well as python distribution. If you're on Linux and installed PyTuning with your system python, there's a good chance it's in `/usr/bin`. If you installed into the Miniconda distribution, then it will probably be somewhere like `~/miniconda/bin`.

Once you've launched the console, this script should be loaded into the environment with the `%load` command. This will load it into the console, but you'll need to execute it. This is normally done with `Shift-Enter`, although `Control-enter` may be used on some platforms/versions.



```
Jupyter QtConsole 4.2.1
Python 3.5.2 |Continuum Analytics, Inc.| (default, Jul  2 2016, 17:53:06)
Type "copyright", "credits" or "license" for more information.

IPython 5.1.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: %load /opt/miniconda/bin/interactive.py
```

This will bring load the script into the console, at which point a `[Shit-Enter]` will execute it.



```
   ...: def euler_fokker():
   ...:     global scale
   ...:     global cents
   ...:     intervals = input("Intervals (list of prime integers): ")
   ...:     octave        = input("Formal Octave (2):")
   ...:
   ...:     if len(octave) == 0:
   ...:         octave = 2
   ...:     else:
   ...:         octave = int(octave)
   ...:
   ...:     exec("int_parsed =%s" % intervals, globals())
   ...:     multiplicities = [1]* len(intervals)
   ...:     scale = create_euler_fokker_scale(int_parsed, multiplicities, octave)
   ...:     print( "Current Scale:")
   ...:     display(scale)
   ...:     cents = [ratio_to_cents(x) for x in scale]
   ...:
   ...: def set_generators():
   ...:     global generators
   ...:     index = 0
   ...:     for entry in all_constructors:
   ...:         print( index, entry[1])
   ...:         index = index + 1
   ...:     selected = input("Which generator?:")
   ...:     generators = all_constructors[int(selected)][0]
   ...:
   ...: Interval = lambda p, q: normalize_interval(sp.Rational(p,q))
```

3.9.3 A (Very) Brief Introduction to Jupyter

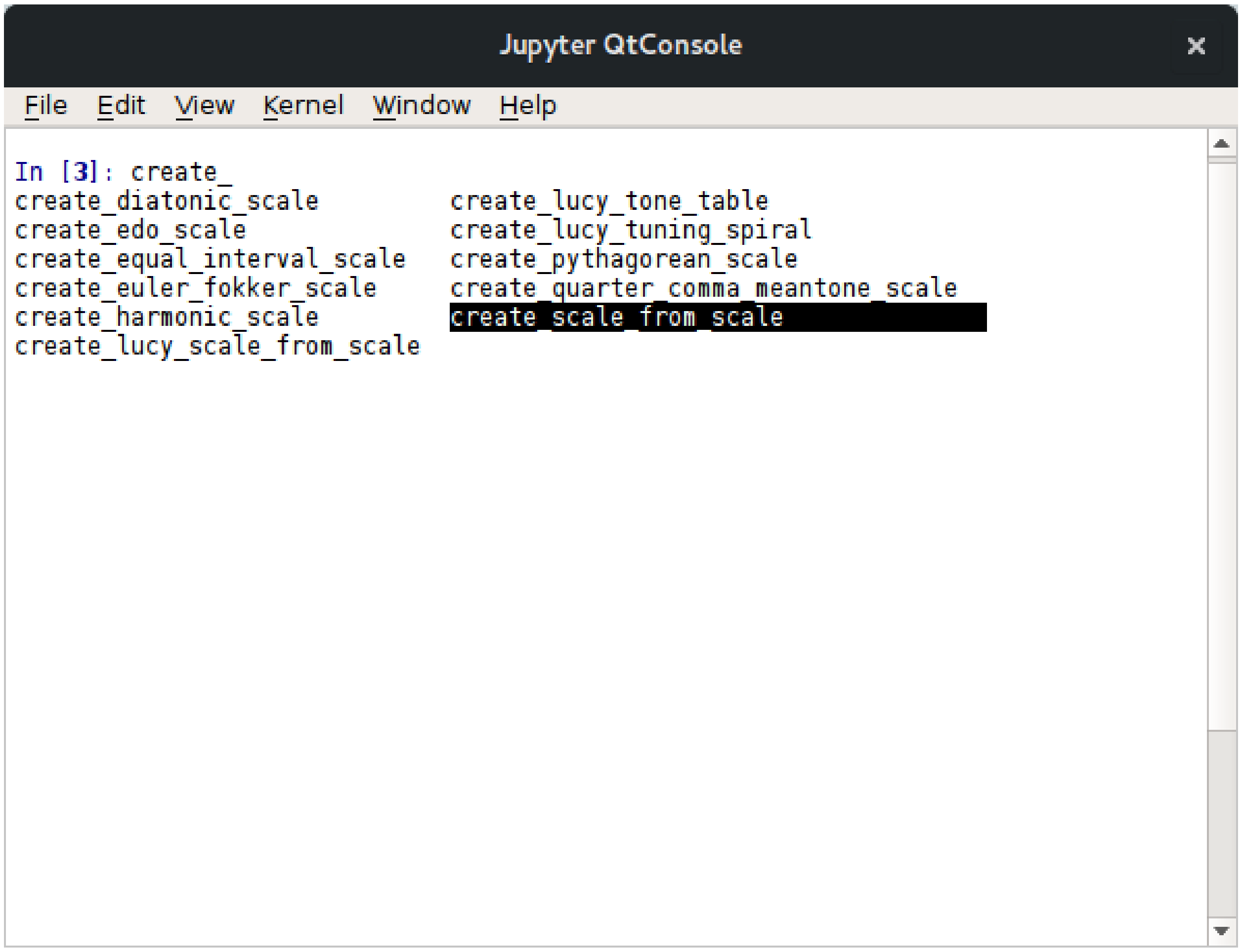
There are a few things about Jupyter which are useful for interacting with the Python interpreter.

Tab Completion

Jupyter has a very good tab completion system which can save a lot of typing.

As a first example, the `%load` command (above) can use completion to navigate to the file. One need only type enough to disambiguate each directory in the path and the tab will complete it, much in the same way that the bash shell will do so.

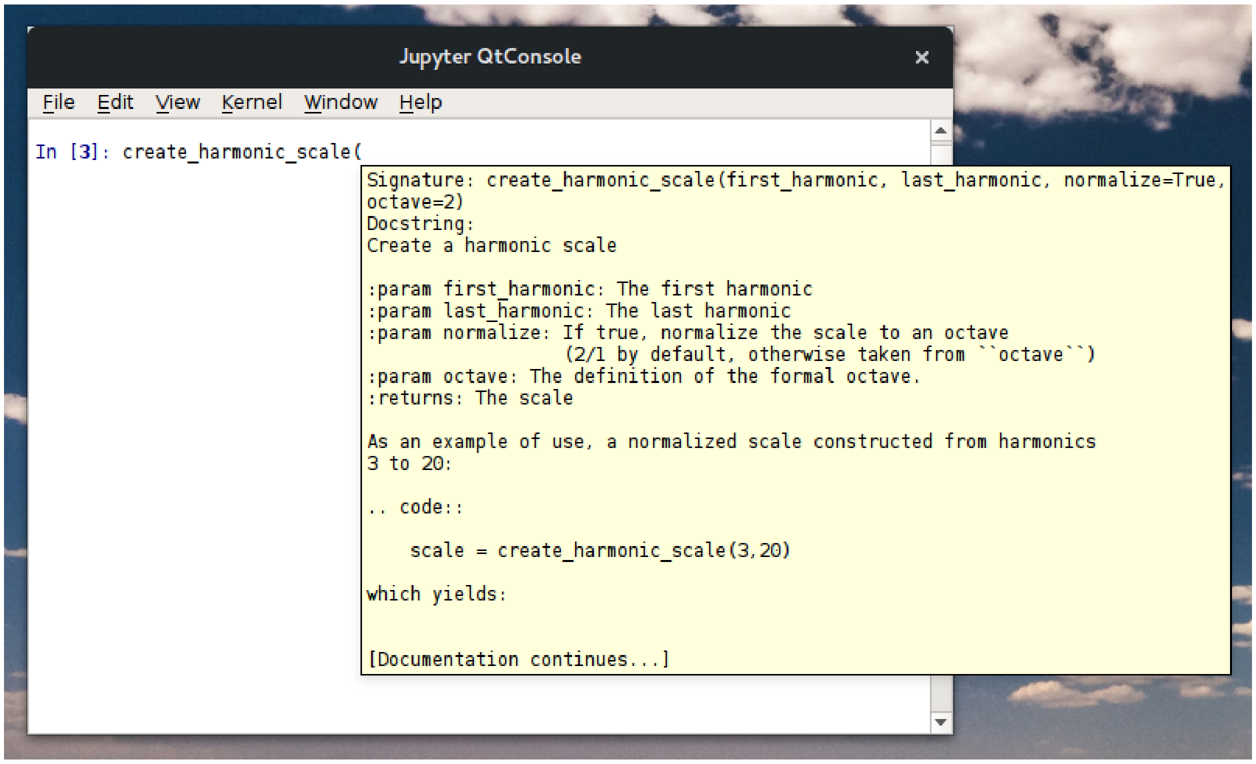
Tab completion can also be used to find functions that are defined in your namespace. As an example, by typing `create_` into the console and hitting tab you will see all objects and functions that begin with that string, and by hitting the tab a second time a selector will be brought up that allows you to select the function you're after:



```
In [3]: create_
create_diatonic_scale        create_lucy_tone_table
create_edo_scale             create_lucy_tuning_spiral
create_equal_interval_scale  create_pythagorean_scale
create_euler_fokker_scale    create_quarter_comma_meantone_scale
create_harmonic_scale        create_scale_from_scale
create_lucy_scale_from_scale
```

Tool Tips

Jupyter also has a nice tool-tip function that will show a function's documentation once the opening (is typed:



```
In [3]: create_harmonic_scale(
Signature: create_harmonic_scale(first_harmonic, last_harmonic, normalize=True,
octave=2)
Docstring:
Create a harmonic scale

:param first_harmonic: The first harmonic
:param last_harmonic: The last harmonic
:param normalize: If true, normalize the scale to an octave
                  (2/1 by default, otherwise taken from ``octave``)
:param octave: The definition of the formal octave.
:returns: The scale

As an example of use, a normalized scale constructed from harmonics
3 to 20:

.. code::

    scale = create_harmonic_scale(3,20)

which yields:


[Documentation continues...]
```

History

Jupyter also has a nice history function. If, for example, at some point in your session you entered the following:

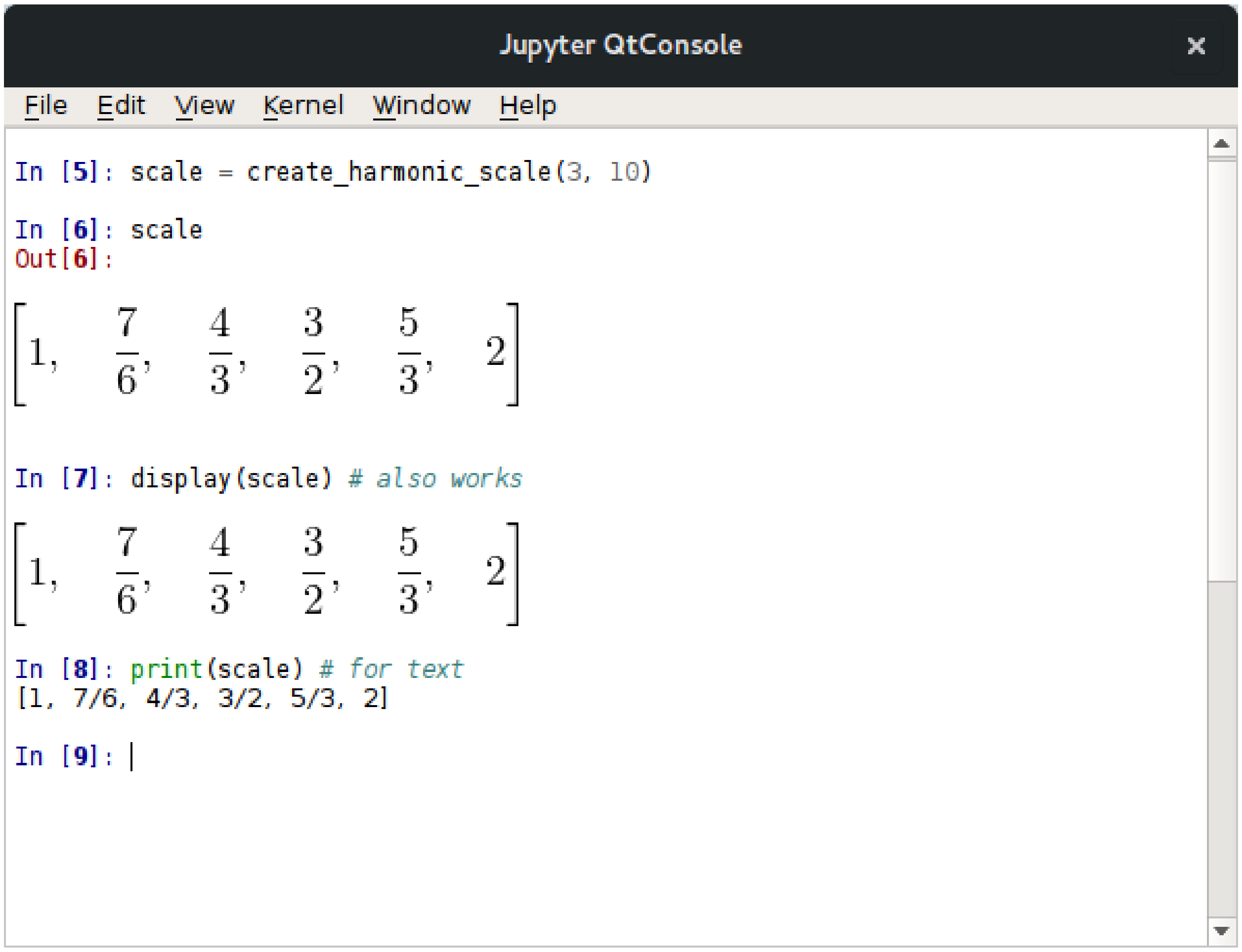
```
In [3]: scale = create_harmonic_scale(2,6)

In [4]:
```

Then, later on in the session if you type `scale =`, each time you hit the up arrow it will search through your history and bring up lines beginning with that character string. You can then edit that line and make changes.

```
In [3]: scale = create_harmonic_scale(2,6)

In [4]: scale2 = create_harmonic_scale(2,7)

In [5]:
```

Rich Display

By default scales and degrees will be displayed symbolically. If you want text display you can use the `print()` function.



```
In [5]: scale = create_harmonic_scale(3, 10)

In [6]: scale
Out[6]:
```

$$\left[1, \frac{7}{6}, \frac{4}{3}, \frac{3}{2}, \frac{5}{3}, 2\right]$$

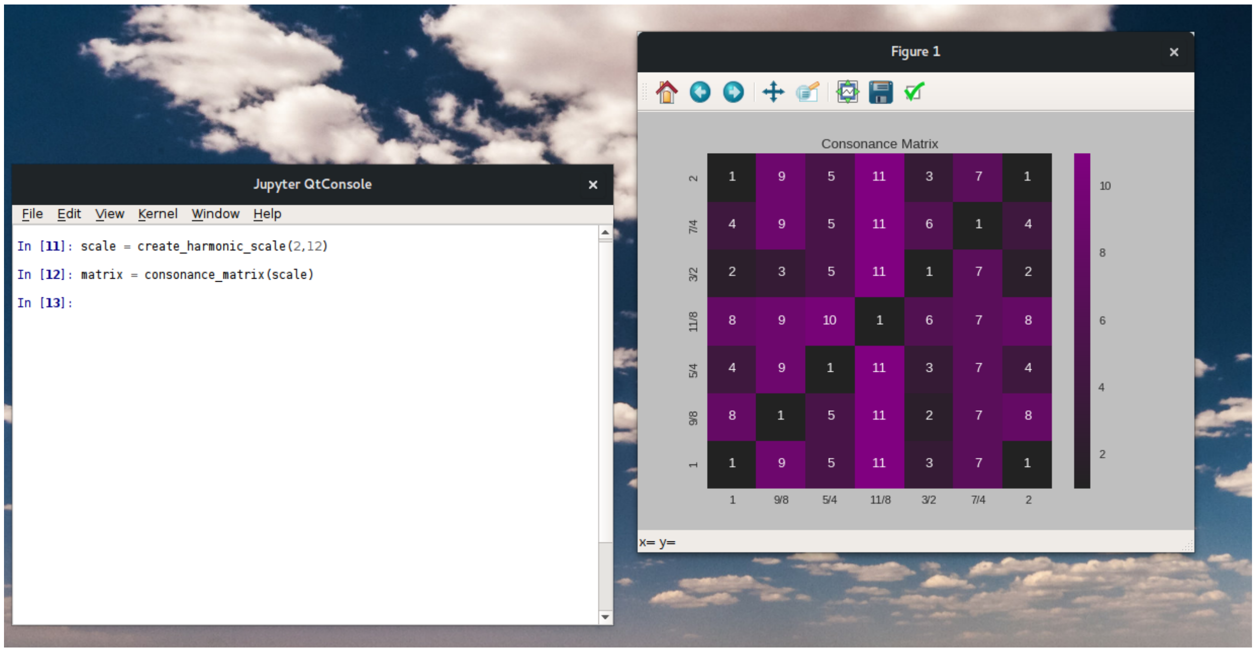
```
In [7]: display(scale) # also works
```

$$\left[1, \frac{7}{6}, \frac{4}{3}, \frac{3}{2}, \frac{5}{3}, 2\right]$$

```
In [8]: print(scale) # for text
[1, 7/6, 4/3, 3/2, 5/3, 2]

In [9]:
```

Graphics

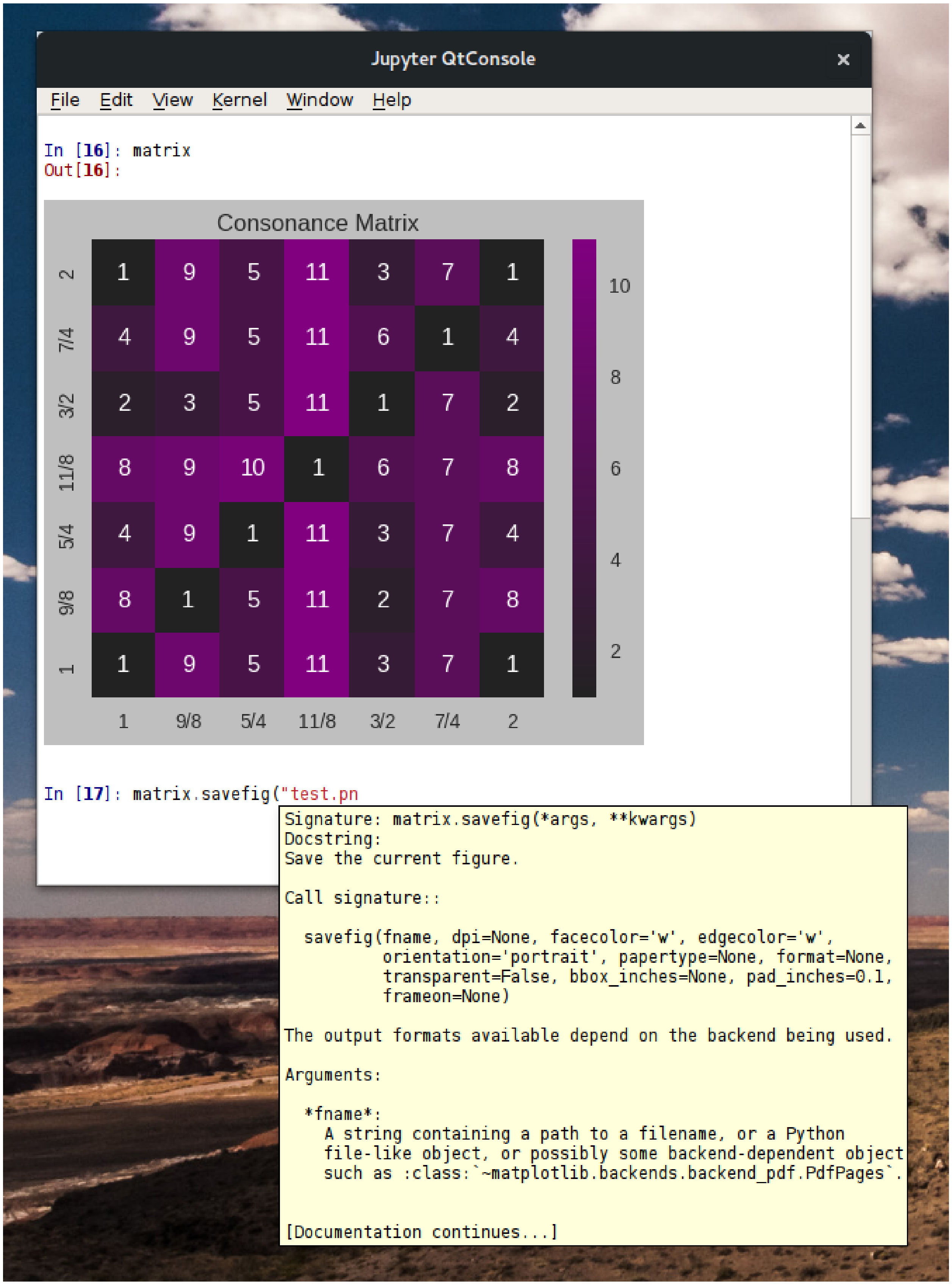
With `matplotlib` installed one also has access to graphics. A graph can be displayed within the console, or it can be displayed in a simple viewer that allows some editing and the saving of the graphic file in a few different formats (JPEG, PNG, SVG). The viewer comes up automatically. If you close it at want to bring it back up later, you can use the `show()` function (i.e., `matrix.show()`).



```
In [11]: scale = create_harmonic_scale(2,12)

In [12]: matrix = consonance_matrix(scale)

In [13]:
```

Note that by re-sizing the window, you re-size the graphic.

You can also save the figure directly from the console:



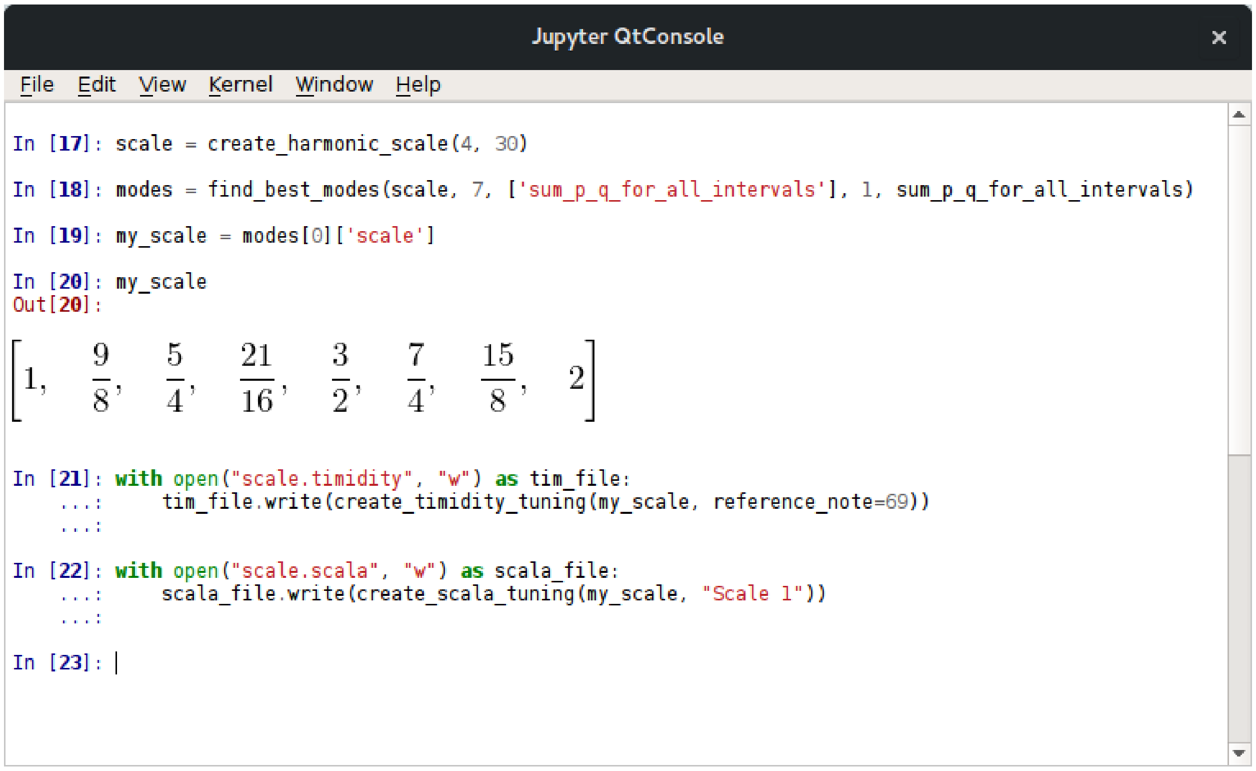
```
In [16]: matrix
Out[16]:
```

```
In [17]: matrix.savefig("test.pn
```

```
Signature: matrix.savefig(*args, **kwargs)
Docstring:
Save the current figure.

Call signature::

  savefig(fname, dpi=None, facecolor='w', edgecolor='w',
          orientation='portrait', papertype=None, format=None,
          transparent=False, bbox_inches=None, pad_inches=0.1,
          frameon=None)

The output formats available depend on the backend being used.

Arguments:

  *fname*:
    A string containing a path to a filename, or a Python
    file-like object, or possibly some backend-dependent object
    such as :class:`~matplotlib.backends.backend_pdf.PdfPages`.


[Documentation continues...]
```

3.9.4 A Sample Session

On my personal website I discuss a scale that I've been working with recently for music composition. It's a mode of the harmonic scale which minimizes dissonance by one of the metrics included in the distribution. In the following session I create the scale and create two tuning tables (a timidity and scala table) for use in music composition.



```
In [17]: scale = create_harmonic_scale(4, 30)

In [18]: modes = find_best_modes(scale, 7, ['sum_p_q_for_all_intervals'], 1, sum_p_q_for_all_intervals)

In [19]: my_scale = modes[0]['scale']

In [20]: my_scale
Out[20]:
```

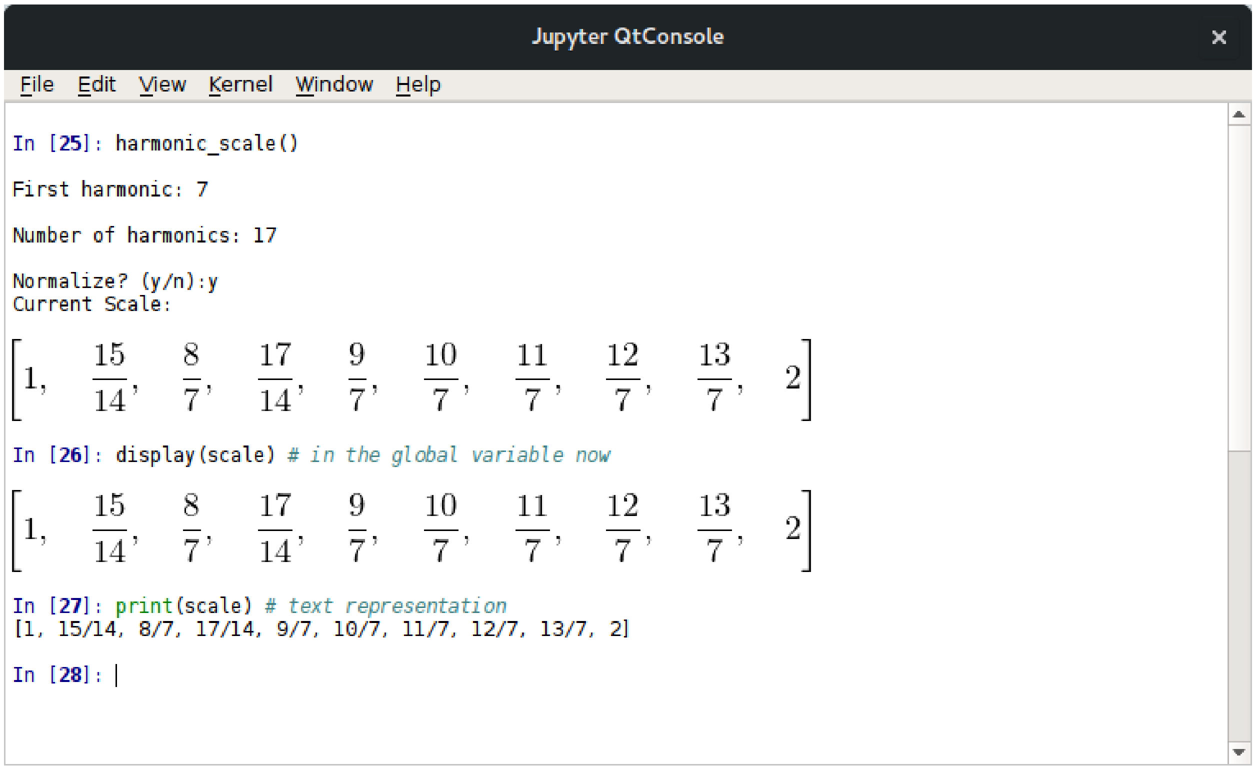
$$\left[1, \frac{9}{8}, \frac{5}{4}, \frac{21}{16}, \frac{3}{2}, \frac{7}{4}, \frac{15}{8}, 2\right]$$

```
In [21]: with open("scale.timidity", "w") as tim_file:
    ...:     tim_file.write(create_timidity_tuning(my_scale, reference_note=69))
    ...:

In [22]: with open("scale.scala", "w") as scala_file:
    ...:     scala_file.write(create_scala_tuning(my_scale, "Scale 1"))
    ...:

In [23]: |
```

3.9.5 Helper Functions

`interactive.py` also creates a few helper functions for the creation of scales. They wrap the base functions in an interactive prompt and define a global variable, `scales` into which the calculated scale is placed.

As an example, to create a harmonic scale:



```
In [25]: harmonic_scale()

First harmonic: 7

Number of harmonics: 17

Normalize? (y/n):y
Current Scale:
```

$$\left[1, \frac{15}{14}, \frac{8}{7}, \frac{17}{14}, \frac{9}{7}, \frac{10}{7}, \frac{11}{7}, \frac{12}{7}, \frac{13}{7}, 2\right]$$

```
In [26]: display(scale) # in the global variable now
```

$$\left[1, \frac{15}{14}, \frac{8}{7}, \frac{17}{14}, \frac{9}{7}, \frac{10}{7}, \frac{11}{7}, \frac{12}{7}, \frac{13}{7}, 2\right]$$

```
In [27]: print(scale) # text representation
[1, 15/14, 8/7, 17/14, 9/7, 10/7, 11/7, 12/7, 13/7, 2]

In [28]: |
```

Only a few functions have yet been written, but more will be included in future releases.

3.9.6 Running in a Persistant Way

Jupyter also offers an interactive notebook, similar to a Matlab notebook. For more complicated analysis it is my preferred way of interacting with the PyTuning library. Documentation, graphics, equations, code, and output calculations can all be included. It can be installed in a way that is similar to the console (and in fact may be installed along with it, depending on how the packages maintainers on your platform have chosen to break things up).

The Github repository for this project has a directory which contains a rendered notebook that shows an exploration of pentatonic scales in the Pythagorean tuning. Github renders notebooks well, so you can see what's possible to decide if you want to install the software. If you're going to be doing anything really complicated in an interactive environment, I would recommend installing and using this.

CHAPTER 4

Index

- genindex

Index