thinkdsp

index
/home/downey/ThinkDSP/code/thinkdsp.py

```
This file contains code used in "Think DSP",
by Allen B. Downey, available from greenteapress.com

Copyright 2013 Allen B. Downey
License: GNU GPLv3 http://www.gnu.org/licenses/gpl.html
```

Modules

array	numpy	scipy	thinkplot
copy	matplotlib.pyplot	struct	warnings
math	random	subprocess	

Classes

- exceptions.Exception(exceptions.BaseException)
  - UnimplementedMethodException
- IntegratedSpectrum
- Signal
  - Chirp
    - ExpoChirp
  - Impulses
  - SilentSignal
  - Sinusoid
    - ComplexSinusoid
    - GlottalSignal
    - ParabolicSignal
      - CubicSignal
    - SawtoothSignal
    - SquareSignal
    - TriangleSignal
  - SumSignal
- Spectrogram
- WavFileWriter
- Wave
- _Noise(Signal)
  - BrownianNoise
  - PinkNoise
  - UncorrelatedGaussianNoise
  - UncorrelatedUniformNoise
- _SpectrumParent
  - Dct
  - Spectrum

class **BrownianNoise**(_Noise)

```
Represents Brownian noise, aka red noise.
```

Method resolution order:

- BrownianNoise
- _Noise
- Signal

Methods defined here:

evaluate(self, ts)

```
Evaluates the signal at the given times.

Computes Brownian noise by taking the cumulative sum of
a uniform random series.

ts: float array of times

returns: float wave array
```

Methods inherited from _Noise:

__init__(self, amp=1.0)

```
Initializes a white noise signal.

amp: float amplitude, 1.0 is nominal max
```

Data descriptors inherited from _Noise:

period

```
Period of the signal in seconds.

returns: float seconds
```

Methods inherited from Signal:

__add__(self, other)

```
Adds two signals.

other: Signal

returns: Signal
```

__radd__ = __add__(self, other)

```
Adds two signals.

other: Signal

returns: Signal
```

make_wave(self, duration=1, start=0, framerate=11025)

```
Makes a Wave object.

duration: float seconds
start: float seconds
framerate: int frames per second

returns: Wave
```

plot(self, framerate=11025)

```
Plots the signal.

The default behavior is to plot three periods.

framerate: samples per second
```

class **Chirp**(Signal)

```
Represents a signal with variable frequency.
```

Methods defined here:

__init__(self, start=440, end=880, amp=1.0)

```
Initializes a linear chirp.

start: float frequency in Hz
end: float frequency in Hz
amp: float amplitude, 1.0 is nominal max
```

evaluate(self, ts)

```
Evaluates the signal at the given times.

ts: float array of times

returns: float wave array
```

Data descriptors defined here:

period

```
Period of the signal in seconds.

returns: float seconds
```

Methods inherited from Signal:

__add__(self, other)

```
Adds two signals.

other: Signal

returns: Signal
```

__radd__ = __add__(self, other)

```
Adds two signals.

other: Signal

returns: Signal
```

make_wave(self, duration=1, start=0, framerate=11025)

```
Makes a Wave object.

duration: float seconds
start: float seconds
framerate: int frames per second

returns: Wave
```

plot(self, framerate=11025)

```
Plots the signal.

The default behavior is to plot three periods.

framerate: samples per second
```

class **ComplexSinusoid**(Sinusoid)

```
Represents a complex exponential signal.
```

Method resolution order:
ComplexSinusoid
Sinusoid
Signal

Methods defined here:

evaluate(self, ts)
```
Evaluates the signal at the given times.

ts: float array of times

returns: float wave array
```

Methods inherited from Sinusoid:

__init__(self, freq=440, amp=1.0, offset=0, func=<ufunc 'sin'>)
```
Initializes a sinusoidal signal.

freq: float frequency in Hz
amp: float amplitude, 1.0 is nominal max
offset: float phase offset in radians
func: function that maps phase to amplitude
```

Data descriptors inherited from Sinusoid:

period
```
Period of the signal in seconds.

returns: float seconds
```

Methods inherited from Signal:

__add__(self, other)
```
Adds two signals.

other: Signal

returns: Signal
```

__radd__ = __add__(self, other)
```
Adds two signals.

other: Signal

returns: Signal
```

make_wave(self, duration=1, start=0, framerate=11025)
```
Makes a Wave object.

duration: float seconds
```

```
start: float seconds
framerate: int frames per second

returns: Wave
```

plot(self, framerate=11025)

```
Plots the signal.

The default behavior is to plot three periods.

framerate: samples per second
```

class **CubicSignal**(ParabolicSignal)

```
Represents a cubic signal.
```

Method resolution order:

- CubicSignal
- ParabolicSignal
- Sinusoid
- Signal

Methods defined here:

evaluate(self, ts)

```
Evaluates the signal at the given times.

ts: float array of times

returns: float wave array
```

Methods inherited from Sinusoid:

__init__(self, freq=440, amp=1.0, offset=0, func=<ufunc 'sin'>)

```
Initializes a sinusoidal signal.

freq: float frequency in Hz
amp: float amplitude, 1.0 is nominal max
offset: float phase offset in radians
func: function that maps phase to amplitude
```

Data descriptors inherited from Sinusoid:

period

```
Period of the signal in seconds.

returns: float seconds
```

Methods inherited from Signal:

__add__(self, other)
```
Adds two signals.

other: Signal

returns: Signal
```

__radd__ = __add__(self, other)
```
Adds two signals.

other: Signal

returns: Signal
```

make_wave(self, duration=1, start=0, framerate=11025)
```
Makes a Wave object.

duration: float seconds
start: float seconds
framerate: int frames per second

returns: Wave
```

plot(self, framerate=11025)
```
Plots the signal.

The default behavior is to plot three periods.

framerate: samples per second
```

class **Dct**(_SpectrumParent)
```
Represents the spectrum of a signal using discrete cosine transform.
```

Methods defined here:

__add__(self, other)
```
Adds two DCTs elementwise.

other: DCT

returns: new DCT
```

__radd__ = __add__(self, other)

make_wave(self)
```
Transforms to the time domain.

returns: Wave
```

Data descriptors defined here:

amps

```
Returns a sequence of amplitudes (read-only property).

Note: for DCTs, amps are positive or negative real.
```

Methods inherited from _SpectrumParent:

__init__(self, hs, fs, framerate, full=False)

```
Initializes a spectrum.

hs: array of amplitudes (real or complex)
fs: array of frequencies
framerate: frames per second
full: boolean to indicate full or real FFT
```

copy(self)

```
Makes a copy.

Returns: new Spectrum
```

estimate_slope(self)

```
Runs linear regression on log power vs log frequency.

returns: slope, inter, r2, p, stderr
```

invert(self)

```
Inverts this spectrum/filter.

returns: new Wave
```

max_diff(self, other)

```
Computes the maximum absolute difference between spectra.

other: Spectrum

returns: float
```

peaks(self)

```
Finds the highest peaks and their frequencies.

returns: sorted list of (amplitude, frequency) pairs
```

plot(self, high=None, **options)

```
Plots amplitude vs frequency.

Note: if this is a full spectrum, it ignores low and high

high: frequency to cut off at
```

plot_power(self, high=None, **options)

```
Plots power vs frequency.

high: frequency to cut off at
```

ratio(self, denom, thresh=1, val=0)

```
The ratio of two spectrums.

denom: Spectrum
thresh: values smaller than this are replaced
val: with this value

returns: new Wave
```

render_full(self, high=None)

```
Extracts amps and fs from a full spectrum.

high: cutoff frequency

returns: fs, amps
```

Data descriptors inherited from _SpectrumParent:

freq_res

max_freq

```
Returns the Nyquist frequency for this spectrum.
```

power

```
Returns a sequence of powers (read-only property).
```

class **ExpoChirp**(Chirp)

```
Represents a signal with varying frequency.
```

Method resolution order:

- ExpoChirp
- Chirp
- Signal

Methods defined here:

evaluate(self, ts)

```
Evaluates the signal at the given times.

ts: float array of times

returns: float wave array
```

Methods inherited from Chirp:

__init__(self, start=440, end=880, amp=1.0)

```
Initializes a linear chirp.
```

```
start: float frequency in Hz
end: float frequency in Hz
amp: float amplitude, 1.0 is nominal max
```

Data descriptors inherited from Chirp:

period

```
Period of the signal in seconds.

returns: float seconds
```

Methods inherited from Signal:

__add__(self, other)

```
Adds two signals.

other: Signal

returns: Signal
```

__radd__ = __add__(self, other)

```
Adds two signals.

other: Signal

returns: Signal
```

make_wave(self, duration=1, start=0, framerate=11025)

```
Makes a Wave object.

duration: float seconds
start: float seconds
framerate: int frames per second

returns: Wave
```

plot(self, framerate=11025)

```
Plots the signal.

The default behavior is to plot three periods.

framerate: samples per second
```

class **GlottalSignal**(Sinusoid)

```
Represents a periodic signal that resembles a glottal signal.
```

Method resolution order:

GlottalSignal
Sinusoid
Signal

Methods defined here:

evaluate(self, ts)

```
Evaluates the signal at the given times.

ts: float array of times

returns: float wave array
```

Methods inherited from Sinusoid:

__init__(self, freq=440, amp=1.0, offset=0, func=<ufunc 'sin'>)

```
Initializes a sinusoidal signal.

freq: float frequency in Hz
amp: float amplitude, 1.0 is nominal max
offset: float phase offset in radians
func: function that maps phase to amplitude
```

Data descriptors inherited from Sinusoid:

period

```
Period of the signal in seconds.

returns: float seconds
```

Methods inherited from Signal:

__add__(self, other)

```
Adds two signals.

other: Signal

returns: Signal
```

__radd__ = __add__(self, other)

```
Adds two signals.

other: Signal

returns: Signal
```

make_wave(self, duration=1, start=0, framerate=11025)

```
Makes a Wave object.

duration: float seconds
start: float seconds
framerate: int frames per second

returns: Wave
```

plot(self, framerate=11025)

```
Plots the signal.

The default behavior is to plot three periods.

framerate: samples per second
```

class **Impulses**(Signal)

```
Represents silence.
```

Methods defined here:

__init__(self, locations, amps=1)

evaluate(self, ts)

```
Evaluates the signal at the given times.

ts: float array of times

returns: float wave array
```

Methods inherited from Signal:

__add__(self, other)

```
Adds two signals.

other: Signal

returns: Signal
```

__radd__ = __add__(self, other)

```
Adds two signals.

other: Signal

returns: Signal
```

make_wave(self, duration=1, start=0, framerate=11025)

```
Makes a Wave object.

duration: float seconds
start: float seconds
framerate: int frames per second

returns: Wave
```

plot(self, framerate=11025)

```
Plots the signal.

The default behavior is to plot three periods.
```

```
framerate: samples per second
```

Data descriptors inherited from Signal:

period

```
Period of the signal in seconds (property).

Since this is used primarily for purposes of plotting,
the default behavior is to return a value, 0.1 seconds,
that is reasonable for many signals.

returns: float seconds
```

class **IntegratedSpectrum**

```
Represents the integral of a spectrum.
```

Methods defined here:

__init__(self, cs, fs)

```
Initializes an integrated spectrum:

cs: sequence of cumulative amplitudes
fs: sequence of frequencies
```

estimate_slope(self, low=1, high=-12000)

```
Runs linear regression on log cumulative power vs log frequency.

returns: slope, inter, r2, p, stderr
```

plot_power(self, low=0, high=None, expo=False, **options)

```
Plots the integrated spectrum.

low: int index to start at
high: int index to end at
```

class **ParabolicSignal**(Sinusoid)

```
Represents a parabolic signal.
```

Method resolution order:

- ParabolicSignal
- Sinusoid
- Signal

Methods defined here:

evaluate(self, ts)

```
Evaluates the signal at the given times.

ts: float array of times

returns: float wave array
```

Methods inherited from Sinusoid:

__init__(self, freq=440, amp=1.0, offset=0, func=<ufunc 'sin'>)

```
Initializes a sinusoidal signal.

freq: float frequency in Hz
amp: float amplitude, 1.0 is nominal max
offset: float phase offset in radians
func: function that maps phase to amplitude
```

Data descriptors inherited from Sinusoid:

period

```
Period of the signal in seconds.

returns: float seconds
```

Methods inherited from Signal:

__add__(self, other)

```
Adds two signals.

other: Signal

returns: Signal
```

__radd__ = __add__(self, other)

```
Adds two signals.

other: Signal

returns: Signal
```

make_wave(self, duration=1, start=0, framerate=11025)

```
Makes a Wave object.

duration: float seconds
start: float seconds
framerate: int frames per second

returns: Wave
```

plot(self, framerate=11025)

```
Plots the signal.

The default behavior is to plot three periods.
```

```
framerate: samples per second
```

class **PinkNoise**(_Noise)

```
Represents Brownian noise, aka red noise.
```

Method resolution order:
PinkNoise
_Noise
Signal

Methods defined here:

__init__(self, amp=1.0, beta=1.0)

```
Initializes a pink noise signal.

amp: float amplitude, 1.0 is nominal max
```

make_wave(self, duration=1, start=0, framerate=11025)

```
Makes a Wave object.

duration: float seconds
start: float seconds
framerate: int frames per second

returns: Wave
```

Data descriptors inherited from _Noise:

period

```
Period of the signal in seconds.

returns: float seconds
```

Methods inherited from Signal:

__add__(self, other)

```
Adds two signals.

other: Signal

returns: Signal
```

__radd__ = __add__(self, other)

```
Adds two signals.

other: Signal

returns: Signal
```

plot(self, framerate=11025)

```
Plots the signal.

The default behavior is to plot three periods.

framerate: samples per second
```

class **SawtoothSignal**(Sinusoid)

```
Represents a sawtooth signal.
```

Method resolution order:

- SawtoothSignal
- Sinusoid
- Signal

Methods defined here:

evaluate(self, ts)

```
Evaluates the signal at the given times.

ts: float array of times

returns: float wave array
```

Methods inherited from Sinusoid:

__init__(self, freq=440, amp=1.0, offset=0, func=<ufunc 'sin'>)

```
Initializes a sinusoidal signal.

freq: float frequency in Hz
amp: float amplitude, 1.0 is nominal max
offset: float phase offset in radians
func: function that maps phase to amplitude
```

Data descriptors inherited from Sinusoid:

period

```
Period of the signal in seconds.

returns: float seconds
```

Methods inherited from Signal:

__add__(self, other)

```
Adds two signals.

other: Signal
```

```
returns: Signal
```

__radd__ = __add__(self, other)

```
Adds two signals.

other: Signal

returns: Signal
```

make_wave(self, duration=1, start=0, framerate=11025)

```
Makes a Wave object.

duration: float seconds
start: float seconds
framerate: int frames per second

returns: Wave
```

plot(self, framerate=11025)

```
Plots the signal.

The default behavior is to plot three periods.

framerate: samples per second
```

class **Signal**

```
Represents a time-varying signal.
```

Methods defined here:

__add__(self, other)

```
Adds two signals.

other: Signal

returns: Signal
```

__radd__ = __add__(self, other)

make_wave(self, duration=1, start=0, framerate=11025)

```
Makes a Wave object.

duration: float seconds
start: float seconds
framerate: int frames per second

returns: Wave
```

plot(self, framerate=11025)

```
Plots the signal.

The default behavior is to plot three periods.
```

```
framerate: samples per second
```

Data descriptors defined here:

period

```
Period of the signal in seconds (property).

Since this is used primarily for purposes of plotting,
the default behavior is to return a value, 0.1 seconds,
that is reasonable for many signals.

returns: float seconds
```

class **SilentSignal**(Signal)

```
Represents silence.
```

Methods defined here:

evaluate(self, ts)

```
Evaluates the signal at the given times.

ts: float array of times

returns: float wave array
```

Methods inherited from Signal:

__add__(self, other)

```
Adds two signals.

other: Signal

returns: Signal
```

__radd__ = __add__(self, other)

```
Adds two signals.

other: Signal

returns: Signal
```

make_wave(self, duration=1, start=0, framerate=11025)

```
Makes a Wave object.

duration: float seconds
start: float seconds
framerate: int frames per second

returns: Wave
```

plot(self, framerate=11025)

```
Plots the signal.

The default behavior is to plot three periods.

framerate: samples per second
```

Data descriptors inherited from Signal:

period

```
Period of the signal in seconds (property).

Since this is used primarily for purposes of plotting,
the default behavior is to return a value, 0.1 seconds,
that is reasonable for many signals.

returns: float seconds
```

class **Sinusoid**(Signal)

```
Represents a sinusoidal signal.
```

Methods defined here:

__init__(self, freq=440, amp=1.0, offset=0, func=<ufunc 'sin'>)

```
Initializes a sinusoidal signal.

freq: float frequency in Hz
amp: float amplitude, 1.0 is nominal max
offset: float phase offset in radians
func: function that maps phase to amplitude
```

evaluate(self, ts)

```
Evaluates the signal at the given times.

ts: float array of times

returns: float wave array
```

Data descriptors defined here:

period

```
Period of the signal in seconds.

returns: float seconds
```

Methods inherited from Signal:

__add__(self, other)

```
Adds two signals.
```

```
other: Signal

returns: Signal
```

__radd__ = __add__(self, other)

```
Adds two signals.

other: Signal

returns: Signal
```

make_wave(self, duration=1, start=0, framerate=11025)

```
Makes a Wave object.

duration: float seconds
start: float seconds
framerate: int frames per second

returns: Wave
```

plot(self, framerate=11025)

```
Plots the signal.

The default behavior is to plot three periods.

framerate: samples per second
```

class **Spectrogram**

```
Represents the spectrum of a signal.
```

Methods defined here:

__init__(self, spec_map, seg_length)

```
Initialize the spectrogram.

spec_map: map from float time to Spectrum
seg_length: number of samples in each segment
```

any_spectrum(self)

```
Returns an arbitrary spectrum from the spectrogram.
```

frequencies(self)

```
Sequence of frequencies.

returns: sequence of float freqencies in Hz.
```

make_wave(self)

```
Inverts the spectrogram and returns a Wave.

returns: Wave
```

plot(self, high=None, **options)

```
Make a pseudocolor plot.

high: highest frequency component to plot
```

times(self)

```
Sorted sequence of times.

returns: sequence of float times in seconds
```

Data descriptors defined here:

freq_res

```
Frequency resolution in Hz.
```

time_res

```
Time resolution in seconds.
```

class **Spectrum**(_SpectrumParent)

```
Represents the spectrum of a signal.
```

Methods defined here:

__add__(self, other)

```
Adds two spectrums elementwise.

other: Spectrum

returns: new Spectrum
```

__len__(self)

```
Length of the spectrum.
```

__mul__(self, other)

```
Multiplies two spectrums elementwise.

other: Spectrum

returns: new Spectrum
```

__radd__ = __add__(self, other)

band_stop(self, low_cutoff, high_cutoff, factor=0)

```
Attenuate frequencies between the cutoffs.

low_cutoff: frequency in Hz
high_cutoff: frequency in Hz
factor: what to multiply the magnitude by
```

convolve(self, other)

```
Convolves two Spectrums.

other: Spectrum

returns: Spectrum
```

differentiate(self)

```
Apply the differentiation filter.

returns: new Spectrum
```

high_pass(self, cutoff, factor=0)

```
Attenuate frequencies below the cutoff.

cutoff: frequency in Hz
factor: what to multiply the magnitude by
```

integrate(self)

```
Apply the integration filter.

returns: new Spectrum
```

low_pass(self, cutoff, factor=0)

```
Attenuate frequencies above the cutoff.

cutoff: frequency in Hz
factor: what to multiply the magnitude by
```

make_integrated_spectrum(self)

```
Makes an integrated spectrum.
```

make_wave(self)

```
Transforms to the time domain.

returns: Wave
```

pink_filter(self, beta=1)

```
Apply a filter that would make white noise pink.

beta: exponent of the pink noise
```

scale(self, factor)

```
Multiplies all elements by the given factor.

factor: what to multiply the magnitude by (could be complex)
```

Data descriptors defined here:

angles

```
Returns a sequence of angles (read-only property).
```

imag

```
Returns the imaginary part of the hs (read-only property).
```

real

```
Returns the real part of the hs (read-only property).
```

Methods inherited from [_SpectrumParent]:

__init__(self, hs, fs, framerate, full=False)

```
Initializes a spectrum.

hs: array of amplitudes (real or complex)
fs: array of frequencies
framerate: frames per second
full: boolean to indicate full or real FFT
```

copy(self)

```
Makes a copy.

Returns: new Spectrum
```

estimate_slope(self)

```
Runs linear regression on log power vs log frequency.

returns: slope, inter, r2, p, stderr
```

invert(self)

```
Inverts this spectrum/filter.

returns: new Wave
```

max_diff(self, other)

```
Computes the maximum absolute difference between spectra.

other: Spectrum

returns: float
```

peaks(self)

```
Finds the highest peaks and their frequencies.

returns: sorted list of (amplitude, frequency) pairs
```

plot(self, high=None, **options)

```
Plots amplitude vs frequency.

Note: if this is a full spectrum, it ignores low and high

high: frequency to cut off at
```

plot_power(self, high=None, **options)

```
Plots power vs frequency.
```

```
high: frequency to cut off at
```

ratio(self, denom, thresh=1, val=0)

```
The ratio of two spectrums.

denom: Spectrum
thresh: values smaller than this are replaced
val: with this value

returns: new Wave
```

render_full(self, high=None)

```
Extracts amps and fs from a full spectrum.

high: cutoff frequency

returns: fs, amps
```

Data descriptors inherited from _SpectrumParent:

amps

```
Returns a sequence of amplitudes (read-only property).
```

freq_res

max_freq

```
Returns the Nyquist frequency for this spectrum.
```

power

```
Returns a sequence of powers (read-only property).
```

class **SquareSignal**(Sinusoid)

```
Represents a square signal.
```

Method resolution order:

- SquareSignal
- Sinusoid
- Signal

Methods defined here:

evaluate(self, ts)

```
Evaluates the signal at the given times.

ts: float array of times

returns: float wave array
```

Methods inherited from Sinusoid:

__init__(self, freq=440, amp=1.0, offset=0, func=<ufunc 'sin'>)

```
Initializes a sinusoidal signal.

freq: float frequency in Hz
amp: float amplitude, 1.0 is nominal max
offset: float phase offset in radians
func: function that maps phase to amplitude
```

Data descriptors inherited from Sinusoid:

period

```
Period of the signal in seconds.

returns: float seconds
```

Methods inherited from Signal:

__add__(self, other)

```
Adds two signals.

other: Signal

returns: Signal
```

__radd__ = __add__(self, other)

```
Adds two signals.

other: Signal

returns: Signal
```

make_wave(self, duration=1, start=0, framerate=11025)

```
Makes a Wave object.

duration: float seconds
start: float seconds
framerate: int frames per second

returns: Wave
```

plot(self, framerate=11025)

```
Plots the signal.

The default behavior is to plot three periods.

framerate: samples per second
```

class **SumSignal**(Signal)

```
Represents the sum of signals.
```

Methods defined here:

__init__(self, *args)

```
Initializes the sum.

args: tuple of signals
```

evaluate(self, ts)

```
Evaluates the signal at the given times.

ts: float array of times

returns: float wave array
```

Data descriptors defined here:

period

```
Period of the signal in seconds.

Note: this is not correct; it's mostly a placekeeper.

But it is correct for a harmonic sequence where all
component frequencies are multiples of the fundamental.

returns: float seconds
```

Methods inherited from Signal:

__add__(self, other)

```
Adds two signals.

other: Signal

returns: Signal
```

__radd__ = __add__(self, other)

```
Adds two signals.

other: Signal

returns: Signal
```

make_wave(self, duration=1, start=0, framerate=11025)

```
Makes a Wave object.

duration: float seconds
start: float seconds
framerate: int frames per second

returns: Wave
```

plot(self, framerate=11025)

```
Plots the signal.

The default behavior is to plot three periods.

framerate: samples per second
```

class **TriangleSignal**(Sinusoid)

```
Represents a triangle signal.
```

Method resolution order:

TriangleSignal
Sinusoid
Signal

Methods defined here:

evaluate(self, ts)

```
Evaluates the signal at the given times.

ts: float array of times

returns: float wave array
```

Methods inherited from Sinusoid:

__init__(self, freq=440, amp=1.0, offset=0, func=<ufunc 'sin'>)

```
Initializes a sinusoidal signal.

freq: float frequency in Hz
amp: float amplitude, 1.0 is nominal max
offset: float phase offset in radians
func: function that maps phase to amplitude
```

Data descriptors inherited from Sinusoid:

period

```
Period of the signal in seconds.

returns: float seconds
```

Methods inherited from Signal:

__add__(self, other)

```
Adds two signals.

other: Signal
```

```
returns: Signal
```

__radd__ = __add__(self, other)

```
Adds two signals.

other: Signal

returns: Signal
```

make_wave(self, duration=1, start=0, framerate=11025)

```
Makes a Wave object.

duration: float seconds
start: float seconds
framerate: int frames per second

returns: Wave
```

plot(self, framerate=11025)

```
Plots the signal.

The default behavior is to plot three periods.

framerate: samples per second
```

class **UncorrelatedGaussianNoise**(_Noise)

```
Represents uncorrelated gaussian noise.
```

Method resolution order:

- UncorrelatedGaussianNoise
- _Noise
- Signal

Methods defined here:

evaluate(self, ts)

```
Evaluates the signal at the given times.

ts: float array of times

returns: float wave array
```

Methods inherited from _Noise:

__init__(self, amp=1.0)

```
Initializes a white noise signal.

amp: float amplitude, 1.0 is nominal max
```

Data descriptors inherited from _Noise:

period

```
Period of the signal in seconds.

returns: float seconds
```

Methods inherited from Signal:

__add__(self, other)

```
Adds two signals.

other: Signal

returns: Signal
```

__radd__ = __add__(self, other)

```
Adds two signals.

other: Signal

returns: Signal
```

make_wave(self, duration=1, start=0, framerate=11025)

```
Makes a Wave object.

duration: float seconds
start: float seconds
framerate: int frames per second

returns: Wave
```

plot(self, framerate=11025)

```
Plots the signal.

The default behavior is to plot three periods.

framerate: samples per second
```

class **UncorrelatedUniformNoise**(_Noise)

```
Represents uncorrelated uniform noise.
```

Method resolution order:

UncorrelatedUniformNoise
_Noise
Signal

Methods defined here:

evaluate(self, ts)

```
Evaluates the signal at the given times.

ts: float array of times

returns: float wave array
```

Methods inherited from _Noise:

__init__(self, amp=1.0)

```
Initializes a white noise signal.

amp: float amplitude, 1.0 is nominal max
```

Data descriptors inherited from _Noise:

period

```
Period of the signal in seconds.

returns: float seconds
```

Methods inherited from Signal:

__add__(self, other)

```
Adds two signals.

other: Signal

returns: Signal
```

__radd__ = __add__(self, other)

```
Adds two signals.

other: Signal

returns: Signal
```

make_wave(self, duration=1, start=0, framerate=11025)

```
Makes a Wave object.

duration: float seconds
start: float seconds
framerate: int frames per second

returns: Wave
```

plot(self, framerate=11025)

```
Plots the signal.

The default behavior is to plot three periods.

framerate: samples per second
```

class **UnimplementedMethodException**(exceptions.Exception)

```
Exception if someone calls a method that should be overridden.
```

Method resolution order:
- UnimplementedMethodException
- exceptions.Exception
- exceptions.BaseException
- __builtin__.object

Data descriptors defined here:

__weakref__

```
list of weak references to the object (if defined)
```

Methods inherited from exceptions.Exception:

__init__(...)

```
x.__init__(...) initializes x; see help(type(x)) for signature
```

Data and other attributes inherited from exceptions.Exception:

__new__ = <built-in method __new__ of type object>

```
T.__new__(S, ...) -> a new object with type S, a subtype of T
```

Methods inherited from exceptions.BaseException:

__delattr__(...)

```
x.__delattr__('name') <==> del x.name
```

__getattribute__(...)

```
x.__getattribute__('name') <==> x.name
```

__getitem__(...)

```
x.__getitem__(y) <==> x[y]
```

__getslice__(...)

```
x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.
```

__reduce__(...)

__repr__(...)

```
x.__repr__() <==> repr(x)
```

__setattr__(...)

```
x.__setattr__('name', value) <==> x.name = value
```

__setstate__(...)

__str__(...)
```
x.__str__() <==> str(x)
```

__unicode__(...)

Data descriptors inherited from exceptions.BaseException:

__dict__

args

message

class **WavFileWriter**

```
Writes wav files.
```

Methods defined here:

__init__(self, filename='sound.wav', framerate=11025)
```
Opens the file and sets parameters.

filename: string
framerate: samples per second
```

close(self, duration=0)
```
Closes the file.

duration: how many seconds of silence to append
```

write(self, wave)
```
Writes a wave.

wave: Wave
```

class **Wave**

```
Represents a discrete-time waveform.
```

Methods defined here:

__add__(self, other)
```
Adds two waves elementwise.

other: Wave

returns: new Wave
```

__init__(self, ys, ts=None, framerate=None)

```
Initializes the wave.

ys: wave array
ts: array of times
framerate: samples per second
```

__len__(self)

__mul__(self, other)

```
Multiplies two waves elementwise.

Note: this operation ignores the timestamps; the result
has the timestamps of self.

other: Wave

returns: new Wave
```

__or__(self, other)

```
Concatenates two waves.

other: Wave

returns: new Wave
```

__radd__ = __add__(self, other)

apodize(self, denom=20, duration=0.1)

```
Tapers the amplitude at the beginning and end of the signal.

Tapers either the given duration of time or the given
fraction of the total duration, whichever is less.

denom: float fraction of the segment to taper
duration: float duration of the taper in seconds
```

convolve(self, other)

```
Convolves two waves.

Note: this operation ignores the timestamps; the result
has the timestamps of self.

other: Wave or NumPy array

returns: Wave
```

copy(self)

```
Makes a copy.

Returns: new Wave
```

corr(self, other)

```
Correlation coefficient two waves.
```

```
other: Wave

returns: float coefficient of correlation
```

cos_cov(self, k)

```
Covariance with a cosine signal.

freq: freq of the cosine signal in Hz

returns: float covariance
```

cos_transform(self)

```
Discrete cosine transform.

returns: list of frequency, cov pairs
```

cov(self, other)

```
Covariance of two unbiased waves.

other: Wave

returns: float
```

cov_mat(self, other)

```
Covariance matrix of two waves.

other: Wave

returns: 2x2 covariance matrix
```

cumsum(self)

```
Computes the cumulative sum of the elements.

returns: new Wave
```

diff(self)

```
Computes the difference between successive elements.

returns: new Wave
```

find_index(self, t)

```
Find the index corresponding to a given time.
```

get_xfactor(self, options)

hamming(self)

```
Apply a Hamming window to the wave.
```

make_audio(self)

```
Makes an IPython Audio object.
```

make_dct(self)

```
Computes the DCT of this wave.
```

make_spectrogram(self, seg_length, win_flag=True)

```
Computes the spectrogram of the wave.

seg_length: number of samples in each segment
win_flag: boolean, whether to apply hamming window to each segment

returns: Spectrogram
```

make_spectrum(self, full=False)

```
Computes the spectrum using FFT.

returns: Spectrum
```

max_diff(self, other)

```
Computes the maximum absolute difference between waves.

other: Wave

returns: float
```

normalize(self, amp=1.0)

```
Normalizes the signal to the given amplitude.

amp: float amplitude
```

play(self, filename='sound.wav')

```
Plays a wave file.

filename: string
```

plot(self, **options)

```
Plots the wave.
```

plot_vlines(self, **options)

```
Plots the wave with vertical lines for samples.
```

quantize(self, bound, dtype)

```
Maps the waveform to quanta.

bound: maximum amplitude
dtype: numpy data type or string

returns: quantized signal
```

roll(self, roll)

```
Rolls this wave by the given number of locations.
```

scale(self, factor)

```
Multplies the wave by a factor.

factor: scale factor
```

segment(self, start=None, duration=None)

```
Extracts a segment.

start: float start time in seconds
duration: float duration in seconds

returns: Wave
```

shift(self, shift)

```
Shifts the wave left or right in time.

shift: float time shift
```

slice(self, i, j)

```
Makes a slice from a Wave.

i: first slice index
j: second slice index
```

truncate(self, n)

```
Trims this wave to the given length.

n: integer index
```

unbias(self)

```
Unbiases the signal.
```

window(self, window)

```
Apply a window to the wave.

window: sequence of multipliers, same length as self.ys
```

write(self, filename='sound.wav')

```
Write a wave file.

filename: string
```

zero_pad(self, n)

```
Trims this wave to the given length.

n: integer index
```

Data descriptors defined here:

duration

```
Duration (property).

returns: float duration in seconds
```

end

start

Functions

CosSignal(freq=440, amp=1.0, offset=0)

```
Makes a cosine Sinusoid.

freq: float frequency in Hz
amp: float amplitude, 1.0 is nominal max
offset: float phase offset in radians

returns: Sinusoid object
```

SinSignal(freq=440, amp=1.0, offset=0)

```
Makes a sine Sinusoid.

freq: float frequency in Hz
amp: float amplitude, 1.0 is nominal max
offset: float phase offset in radians

returns: Sinusoid object
```

Sinc(freq=440, amp=1.0, offset=0)

```
Makes a Sinc function.

freq: float frequency in Hz
amp: float amplitude, 1.0 is nominal max
offset: float phase offset in radians

returns: Sinusoid object
```

apodize(ys, framerate, denom=20, duration=0.1)

```
Tapers the amplitude at the beginning and end of the signal.

Tapers either the given duration of time or the given
fraction of the total duration, whichever is less.

ys: wave array
framerate: int frames per second
denom: float fraction of the segment to taper
duration: float duration of the taper in seconds

returns: wave array
```

cos_wave(freq, duration=1, offset=0)

```
Makes a cosine wave with the given parameters.

freq: float cycles per second
duration: float seconds
offset: float radians

returns: Wave
```

find_index(x, xs)

```
Find the index corresponding to a given value in an array.
```

infer_framerate(ts)

```
Given ts, find the framerate.

Assumes that the ts are equally spaced.

ts: sequence of times in seconds

returns: frames per second
```

mag(a)

```
Computes the magnitude of a numpy array.

a: numpy array

returns: float
```

main()

make_chord(midi_nums, duration, sig_cons=<function CosSignal>, framerate=11025)

```
Make a chord with the given duration.

midi_nums: sequence of int MIDI note numbers
duration: float seconds
sig_cons: Signal constructor function
framerate: int frames per second

returns: Wave
```

make_note(midi_num, duration, sig_cons=<function CosSignal>, framerate=11025)

```
Make a MIDI note with the given duration.

midi_num: int MIDI note number
duration: float seconds
sig_cons: Signal constructor function
framerate: int frames per second

returns: Wave
```

midi_to_freq(midi_num)

```
Converts MIDI note number to frequency.

midi_num: int MIDI note number

returns: float frequency in Hz
```

normalize(ys, amp=1.0)

```
Normalizes a wave array so the maximum amplitude is +amp or -amp.

ys: wave array
amp: max amplitude (pos or neg) in result

returns: wave array
```

play_wave(filename='sound.wav', player='aplay')

```
Plays a wave file.

filename: string
player: string name of executable that plays wav files
```

quantize(ys, bound, dtype)

```
Maps the waveform to quanta.

ys: wave array
bound: maximum amplitude
dtype: numpy data type of the result

returns: quantized signal
```

random_seed(x)

```
Initialize the random and np.random generators.

x: int seed
```

read_wave(filename='sound.wav')

```
Reads a wave file.

filename: string

returns: Wave
```

rest(duration)

```
Makes a rest of the given duration.

duration: float seconds

returns: Wave
```

shift_left(ys, shift)

```
Shifts a wave array to the left.

ys: wave array
shift: integer shift

returns: wave array
```

shift_right(ys, shift)

```
Shifts a wave array to the right and zero pads.

ys: wave array
shift: integer shift

returns: wave array
```

sin_wave(freq, duration=1, offset=0)

```
Makes a sine wave with the given parameters.

freq: float cycles per second
duration: float seconds
```

```
offset: float radians

returns: Wave
```

truncate(ys, n)

```
Trims a wave array to the given length.

ys: wave array
n: integer length

returns: wave array
```

unbias(ys)

```
Shifts a wave array so it has mean 0.

ys: wave array

returns: wave array
```

zero_pad(array, n)

```
Extends an array with zeros.

array: numpy array
n: length of result

returns: new NumPy array
```

Data

PI2 = 6.283185307179586
division = _Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
print_function = _Feature((2, 6, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 65536)