--- === ---

1 README.md

pdf

2 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
"""

from __future__ import print_function, division

import array
import copy
import math

import numpy as np
import random
import scipy
import scipy.stats
import scipy.fftpack
import struct
import subprocess
import thinkplot
import warnings

from fractions import gcd
from wave import open as open_wave

import matplotlib.pyplot as pyplot

try:
    from IPython.display import Audio
except:
    warnings.warn("Can't import Audio from IPython.display; "
                  "Wave.make_audio() will not work.")

PI2 = math.pi * 2
```

```
def random_seed(x):
    """Initialize the random and np.random generators.

    x: int seed
    """
    random.seed(x)
    np.random.seed(x)


class UnimplementedMethodException(Exception):
    """Exception if someone calls a method that should be
overridden."""


class WavFileWriter:
    """Writes wav files."""

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

        filename: string
        framerate: samples per second
        """
        self.filename = filename
        self.framerate = framerate
        self.nchannels = 1
        self.sampwidth = 2
        self.bits = self.sampwidth * 8
        self.bound = 2**(self.bits-1) - 1

        self.fmt = 'h'
        self.dtype = np.int16

        self.fp = open_wave(self.filename, 'w')
        self.fp.setnchannels(self.nchannels)
        self.fp.setsampwidth(self.sampwidth)
        self.fp.setframerate(self.framerate)

    def write(self, wave):
        """Writes a wave.

        wave: Wave
        """
        zs = wave.quantize(self.bound, self.dtype)
        self.fp.writeframes(zs.tostring())

    def close(self, duration=0):
        """Closes the file.

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

```
        """
        if duration:
            self.write(rest(duration))

        self.fp.close()


def read_wave(filename='sound.wav'):
    """Reads a wave file.

    filename: string

    returns: Wave
    """
    fp = open_wave(filename, 'r')

    nchannels = fp.getnchannels()
    nframes = fp.getnframes()
    sampwidth = fp.getsampwidth()
    framerate = fp.getframerate()

    z_str = fp.readframes(nframes)

    fp.close()

    dtype_map = {1:np.int8, 2:np.int16, 3:'special',
4:np.int32}
    if sampwidth not in dtype_map:
        raise ValueError('sampwidth %d unknown' % sampwidth)

    if sampwidth == 3:
        xs = np.fromstring(z_str,
dtype=np.int8).astype(np.int32)
        ys = (xs[2::3] * 256 + xs[1::3]) * 256 + xs[0::3]
    else:
        ys = np.fromstring(z_str, dtype=dtype_map[sampwidth])

    # if it's in stereo, just pull out the first channel
    if nchannels == 2:
        ys = ys[::2]

    #ts = np.arange(len(ys)) / framerate
    wave = Wave(ys, framerate=framerate)
    wave.normalize()
    return wave


def play_wave(filename='sound.wav', player='aplay'):
    """Plays a wave file.
```

```
    filename: string
    player: string name of executable that plays wav files
    """
    cmd = '%s %s' % (player, filename)
    popen = subprocess.Popen(cmd, shell=True)
    popen.communicate()


def find_index(x, xs):
    """Find the index corresponding to a given value in an
array."""
    n = len(xs)
    start = xs[0]
    end = xs[-1]
    i = round((n-1) * (x - start) / (end - start))
    return int(i)


class _SpectrumParent:
    """Contains code common to Spectrum and DCT.
    """

    def __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
        """
        self.hs = np.asanyarray(hs)
        self.fs = np.asanyarray(fs)
        self.framerate = framerate
        self.full = full

    @property
    def max_freq(self):
        """Returns the Nyquist frequency for this spectrum."""
        return self.framerate / 2

    @property
    def amps(self):
        """Returns a sequence of amplitudes (read-only
property)."""
        return np.absolute(self.hs)

    @property
    def power(self):
        """Returns a sequence of powers (read-only
property)."""
```

```
        return self.amps ** 2

    def copy(self):
        """Makes a copy.

        Returns: new Spectrum
        """
        return copy.deepcopy(self)

    def max_diff(self, other):
        """Computes the maximum absolute difference between
spectra.

        other: Spectrum

        returns: float
        """
        assert self.framerate == other.framerate
        assert len(self) == len(other)

        hs = self.hs - other.hs
        return np.max(np.abs(hs))

    def 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
        """
        ratio_spectrum = self.copy()
        ratio_spectrum.hs /= denom.hs
        ratio_spectrum.hs[denom.amps < thresh] = val
        return ratio_spectrum

    def invert(self):
        """Inverts this spectrum/filter.

        returns: new Wave
        """
        inverse = self.copy()
        inverse.hs = 1 / inverse.hs
        return inverse

    @property
    def freq_res(self):
        return self.framerate / 2 / (len(self.fs) - 1)
```

```
    def render_full(self, high=None):
        """Extracts amps and fs from a full spectrum.

        high: cutoff frequency

        returns: fs, amps
        """
        hs = np.fft.fftshift(self.hs)
        amps = np.abs(hs)
        fs = np.fft.fftshift(self.fs)
        i = 0 if high is None else find_index(-high, fs)
        j = None if high is None else find_index(high, fs) + 1
        return fs[i:j], amps[i:j]

    def 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
        """
        if self.full:
            fs, amps = self.render_full(high)
            thinkplot.plot(fs, amps, **options)
        else:
            i = None if high is None else find_index(high,
self.fs)
            thinkplot.plot(self.fs[:i], self.amps[:i],
**options)

    def plot_power(self, high=None, **options):
        """Plots power vs frequency.

        high: frequency to cut off at
        """
        if self.full:
            fs, amps = self.render_full(high)
            thinkplot.plot(fs, amps**2, **options)
        else:
            i = None if high is None else find_index(high,
self.fs)
            thinkplot.plot(self.fs[:i], self.power[:i],
**options)

    def estimate_slope(self):
        """Runs linear regression on log power vs log
frequency.

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

```
        """
        x = np.log(self.fs[1:])
        y = np.log(self.power[1:])
        t = scipy.stats.linregress(x,y)
        return t

    def peaks(self):
        """Finds the highest peaks and their frequencies.

        returns: sorted list of (amplitude, frequency) pairs
        """
        t = list(zip(self.amps, self.fs))
        t.sort(reverse=True)
        return t


class Spectrum(_SpectrumParent):
    """Represents the spectrum of a signal."""

    def __len__(self):
        """Length of the spectrum."""
        return len(self.hs)

    def __add__(self, other):
        """Adds two spectrums elementwise.

        other: Spectrum

        returns: new Spectrum
        """
        if other == 0:
            return self.copy()

        assert all(self.fs == other.fs)
        hs = self.hs + other.hs
        return Spectrum(hs, self.fs, self.framerate,
self.full)

    __radd__ = __add__

    def __mul__(self, other):
        """Multiplies two spectrums elementwise.

        other: Spectrum

        returns: new Spectrum
        """
        assert all(self.fs == other.fs)
        hs = self.hs * other.hs
        return Spectrum(hs, self.fs, self.framerate,
```

```
self.full)

    def convolve(self, other):
        """Convolves two Spectrums.

        other: Spectrum

        returns: Spectrum
        """
        assert all(self.fs == other.fs)
        if self.full:
            hs1 = np.fft.fftshift(self.hs)
            hs2 = np.fft.fftshift(other.hs)
            hs = np.convolve(hs1, hs2, mode='same')
            hs = np.fft.ifftshift(hs)
        else:
            # not sure this branch would mean very much
            hs = np.convolve(self.hs, other.hs, mode='same')

        return Spectrum(hs, self.fs, self.framerate,
self.full)

    @property
    def real(self):
        """Returns the real part of the hs (read-only
property)."""
        return np.real(self.hs)

    @property
    def imag(self):
        """Returns the imaginary part of the hs (read-only
property)."""
        return np.imag(self.hs)

    @property
    def angles(self):
        """Returns a sequence of angles (read-only
property)."""
        return np.angle(self.hs)

    def scale(self, factor):
        """Multiplies all elements by the given factor.

        factor: what to multiply the magnitude by (could be
complex)
        """
        self.hs *= factor

    def low_pass(self, cutoff, factor=0):
        """Attenuate frequencies above the cutoff.
```

```
        cutoff: frequency in Hz
        factor: what to multiply the magnitude by
        """
        self.hs[abs(self.fs) > cutoff] *= factor

    def high_pass(self, cutoff, factor=0):
        """Attenuate frequencies below the cutoff.

        cutoff: frequency in Hz
        factor: what to multiply the magnitude by
        """
        self.hs[abs(self.fs) < cutoff] *= factor

    def 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
        """
        # TODO: test this function
        fs = abs(self.fs)
        indices = (low_cutoff < fs) & (fs < high_cutoff)
        self.hs[indices] *= factor

    def pink_filter(self, beta=1):
        """Apply a filter that would make white noise pink.

        beta: exponent of the pink noise
        """
        denom = self.fs ** (beta/2.0)
        denom[0] = 1
        self.hs /= denom

    def differentiate(self):
        """Apply the differentiation filter.

        returns: new Spectrum
        """
        new = self.copy()
        new.hs *= PI2 * 1j * new.fs
        return new

    def integrate(self):
        """Apply the integration filter.

        returns: new Spectrum
        """
        new = self.copy()
```

```
        new.hs /= PI2 * 1j * new.fs
        return new

    def make_integrated_spectrum(self):
        """Makes an integrated spectrum.
        """
        cs = np.cumsum(self.power)
        cs /= cs[-1]
        return IntegratedSpectrum(cs, self.fs)

    def make_wave(self):
        """Transforms to the time domain.

        returns: Wave
        """
        if self.full:
            ys = np.fft.ifft(self.hs)
        else:
            ys = np.fft.irfft(self.hs)

        #NOTE: whatever the start time was, we lose it when
        # we transform back; we could fix that by saving start
        # time in the Spectrum
        # ts = self.start + np.arange(len(ys)) /
self.framerate
        return Wave(ys, framerate=self.framerate)


class IntegratedSpectrum:
    """Represents the integral of a spectrum."""

    def __init__(self, cs, fs):
        """Initializes an integrated spectrum:

        cs: sequence of cumulative amplitudes
        fs: sequence of frequencies
        """
        self.cs = np.asanyarray(cs)
        self.fs = np.asanyarray(fs)

    def 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
        """
        cs = self.cs[low:high]
        fs = self.fs[low:high]
```

```
        if expo:
            cs = np.exp(cs)

        thinkplot.plot(fs, cs, **options)

    def estimate_slope(self, low=1, high=-12000):
        """Runs linear regression on log cumulative power vs
log frequency.

        returns: slope, inter, r2, p, stderr
        """
        #print self.fs[low:high]
        #print self.cs[low:high]
        x = np.log(self.fs[low:high])
        y = np.log(self.cs[low:high])
        t = scipy.stats.linregress(x,y)
        return t


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

    @property
    def amps(self):
        """Returns a sequence of amplitudes (read-only
property).

        Note: for DCTs, amps are positive or negative real.
        """
        return self.hs

    def __add__(self, other):
        """Adds two DCTs elementwise.

        other: DCT

        returns: new DCT
        """
        if other == 0:
            return self

        assert self.framerate == other.framerate
        hs = self.hs + other.hs
        return Dct(hs, self.fs, self.framerate)

    __radd__ = __add__

    def make_wave(self):
        """Transforms to the time domain.
```

```
        returns: Wave
        """
        N = len(self.hs)
        ys = scipy.fftpack.idct(self.hs, type=2) / 2 / N
        #NOTE: whatever the start time was, we lose it when
        # we transform back
        #ts = self.start + np.arange(len(ys)) / self.framerate
        return Wave(ys, framerate=self.framerate)


class Spectrogram:
    """Represents the spectrum of a signal."""

    def __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
        """
        self.spec_map = spec_map
        self.seg_length = seg_length

    def any_spectrum(self):
        """Returns an arbitrary spectrum from the
spectrogram."""
        index = next(iter(self.spec_map))
        return self.spec_map[index]

    @property
    def time_res(self):
        """Time resolution in seconds."""
        spectrum = self.any_spectrum()
        return float(self.seg_length) / spectrum.framerate

    @property
    def freq_res(self):
        """Frequency resolution in Hz."""
        return self.any_spectrum().freq_res

    def times(self):
        """Sorted sequence of times.

        returns: sequence of float times in seconds
        """
        ts = sorted(iter(self.spec_map))
        return ts

    def frequencies(self):
        """Sequence of frequencies.
```

```
        returns: sequence of float freqencies in Hz.
        """
        fs = self.any_spectrum().fs
        return fs

    def plot(self, high=None, **options):
        """Make a pseudocolor plot.

        high: highest frequency component to plot
        """
        fs = self.frequencies()
        i = None if high is None else find_index(high, fs)
        fs = fs[:i]
        ts = self.times()

        # make the array
        size = len(fs), len(ts)
        array = np.zeros(size, dtype=np.float)

        # copy amplitude from each spectrum into a column of
the array
        for j, t in enumerate(ts):
            spectrum = self.spec_map[t]
            array[:, j] = spectrum.amps[:i]

        thinkplot.pcolor(ts, fs, array, **options)

    def make_wave(self):
        """Inverts the spectrogram and returns a Wave.

        returns: Wave
        """
        res = []
        for t, spectrum in sorted(self.spec_map.items()):
            wave = spectrum.make_wave()
            n = len(wave)

            window = 1 / np.hamming(n)
            wave.window(window)

            i = wave.find_index(t)
            start = i - n // 2
            end = start + n
            res.append((start, end, wave))

        starts, ends, waves = zip(*res)
        low = min(starts)
        high = max(ends)
```

```
        ys = np.zeros(high-low, np.float)
        for start, end, wave in res:
            ys[start:end] = wave.ys

        # ts = np.arange(len(ys)) / self.framerate
        return Wave(ys, framerate=wave.framerate)


class Wave:
    """Represents a discrete-time waveform.

    """
    def __init__(self, ys, ts=None, framerate=None):
        """Initializes the wave.

        ys: wave array
        ts: array of times
        framerate: samples per second
        """
        self.ys = np.asanyarray(ys)
        self.framerate = framerate if framerate is not None
else 11025

        if ts is None:
            self.ts = np.arange(len(ys)) / self.framerate
        else:
            self.ts = np.asanyarray(ts)

    def copy(self):
        """Makes a copy.

        Returns: new Wave
        """
        return copy.deepcopy(self)

    def __len__(self):
        return len(self.ys)

    @property
    def start(self):
        return self.ts[0]

    @property
    def end(self):
        return self.ts[-1]

    @property
    def duration(self):
        """Duration (property).
```

```
        returns: float duration in seconds
        """
        return len(self.ys) / self.framerate

    def __add__(self, other):
        """Adds two waves elementwise.

        other: Wave

        returns: new Wave
        """
        if other == 0:
            return self

        assert self.framerate == other.framerate

        # make an array of times that covers both waves
        start = min(self.start, other.start)
        end = max(self.end, other.end)
        n = int(round((end - start) * self.framerate)) + 1
        ys = np.zeros(n)
        ts = start + np.arange(n) / self.framerate

        def add_ys(wave):
            i = find_index(wave.start, ts)

            # make sure the arrays line up reasonably well
            diff = ts[i] - wave.start
            dt = 1 / wave.framerate
            if (diff / dt) > 0.1:
                warnings.warn("Can't add these waveforms;
their "
                              "time arrays don't line up.")

            j = i + len(wave)
            ys[i:j] += wave.ys

        add_ys(self)
        add_ys(other)

        return Wave(ys, ts, self.framerate)

    __radd__ = __add__

    def __or__(self, other):
        """Concatenates two waves.

        other: Wave

        returns: new Wave
```

```
        """
        if self.framerate != other.framerate:
            raise ValueError('Wave.__or__: framerates do not
agree')

        ys = np.concatenate((self.ys, other.ys))
        # ts = np.arange(len(ys)) / self.framerate
        return Wave(ys, framerate=self.framerate)

    def __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
        """
        # the spectrums have to have the same framerate and
duration
        assert self.framerate == other.framerate
        assert len(self) == len(other)

        ys = self.ys * other.ys
        return Wave(ys, self.ts, self.framerate)

    def max_diff(self, other):
        """Computes the maximum absolute difference between
waves.

        other: Wave

        returns: float
        """
        assert self.framerate == other.framerate
        assert len(self) == len(other)

        ys = self.ys - other.ys
        return np.max(np.abs(ys))

    def 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
        """
        if isinstance(other, Wave):
            assert self.framerate == other.framerate
            window = other.ys
        else:
            window = other

        ys = np.convolve(self.ys, window, mode='full')
        #ts = np.arange(len(ys)) / self.framerate
        return Wave(ys, framerate=self.framerate)

    def diff(self):
        """Computes the difference between successive
elements.

        returns: new Wave
        """
        ys = np.diff(self.ys)
        ts = self.ts[1:].copy()
        return Wave(ys, ts, self.framerate)

    def cumsum(self):
        """Computes the cumulative sum of the elements.

        returns: new Wave
        """
        ys = np.cumsum(self.ys)
        ts = self.ts.copy()
        return Wave(ys, ts, self.framerate)

    def quantize(self, bound, dtype):
        """Maps the waveform to quanta.

        bound: maximum amplitude
        dtype: numpy data type or string

        returns: quantized signal
        """
        return quantize(self.ys, bound, dtype)

    def 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
        """
        self.ys = apodize(self.ys, self.framerate, denom,
duration)

    def hamming(self):
        """Apply a Hamming window to the wave.
        """
        self.ys *= np.hamming(len(self.ys))

    def window(self, window):
        """Apply a window to the wave.

        window: sequence of multipliers, same length as
self.ys
        """
        self.ys *= window

    def scale(self, factor):
        """Multplies the wave by a factor.

        factor: scale factor
        """
        self.ys *= factor

    def shift(self, shift):
        """Shifts the wave left or right in time.

        shift: float time shift
        """
        # TODO: track down other uses of this function and
check them
        self.ts += shift

    def roll(self, roll):
        """Rolls this wave by the given number of locations.
        """
        self.ys = np.roll(self.ys, roll)

    def truncate(self, n):
        """Trims this wave to the given length.

        n: integer index
        """
        self.ys = truncate(self.ys, n)
        self.ts = truncate(self.ts, n)

    def zero_pad(self, n):
        """Trims this wave to the given length.
```

```
        n: integer index
        """
        self.ys = zero_pad(self.ys, n)
        self.ts = self.start + np.arange(n) / self.framerate

    def normalize(self, amp=1.0):
        """Normalizes the signal to the given amplitude.

        amp: float amplitude
        """
        self.ys = normalize(self.ys, amp=amp)

    def unbias(self):
        """Unbiases the signal.
        """
        self.ys = unbias(self.ys)

    def find_index(self, t):
        """Find the index corresponding to a given time."""
        n = len(self)
        start = self.start
        end = self.end
        i = round((n-1) * (t - start) / (end - start))
        return int(i)

    def segment(self, start=None, duration=None):
        """Extracts a segment.

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

        returns: Wave
        """
        if start is None:
            start = self.ts[0]
            i = 0
        else:
            i = self.find_index(start)

        j = None if duration is None else
self.find_index(start + duration)
        return self.slice(i, j)

    def slice(self, i, j):
        """Makes a slice from a Wave.

        i: first slice index
        j: second slice index
        """
        ys = self.ys[i:j].copy()
```

```
        ts = self.ts[i:j].copy()
        return Wave(ys, ts, self.framerate)

    def make_spectrum(self, full=False):
        """Computes the spectrum using FFT.

        returns: Spectrum
        """
        n = len(self.ys)
        d = 1 / self.framerate

        if full:
            hs = np.fft.fft(self.ys)
            fs = np.fft.fftfreq(n, d)
        else:
            hs = np.fft.rfft(self.ys)
            fs = np.fft.rfftfreq(n, d)

        return Spectrum(hs, fs, self.framerate, full)

    def make_dct(self):
        """Computes the DCT of this wave.
        """
        N = len(self.ys)
        hs = scipy.fftpack.dct(self.ys, type=2)
        fs = (0.5 + np.arange(N)) / 2
        return Dct(hs, fs, self.framerate)

    def 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
        """
        if win_flag:
            window = np.hamming(seg_length)
        i, j = 0, seg_length
        step = int(seg_length // 2)

        # map from time to Spectrum
        spec_map = {}

        while j < len(self.ys):
            segment = self.slice(i, j)
            if win_flag:
                segment.window(window)
```

```
            # the nominal time for this segment is the
midpoint
            t = (segment.start + segment.end) / 2
            spec_map[t] = segment.make_spectrum()

            i += step
            j += step

        return Spectrogram(spec_map, seg_length)

    def get_xfactor(self, options):
        try:
            xfactor = options['xfactor']
            options.pop('xfactor')
        except KeyError:
            xfactor = 1
        return xfactor

    def plot(self, **options):
        """Plots the wave.

        """
        xfactor = self.get_xfactor(options)
        thinkplot.plot(self.ts * xfactor, self.ys, **options)

    def plot_vlines(self, **options):
        """Plots the wave with vertical lines for samples.

        """
        xfactor = self.get_xfactor(options)
        thinkplot.vlines(self.ts * xfactor, 0, self.ys,
**options)

    def corr(self, other):
        """Correlation coefficient two waves.

        other: Wave

        returns: float coefficient of correlation
        """
        corr = np.corrcoef(self.ys, other.ys)[0, 1]
        return corr

    def cov_mat(self, other):
        """Covariance matrix of two waves.

        other: Wave

        returns: 2x2 covariance matrix
        """
```

```
        return np.cov(self.ys, other.ys)

    def cov(self, other):
        """Covariance of two unbiased waves.

        other: Wave

        returns: float
        """
        total = sum(self.ys * other.ys) / len(self.ys)
        return total

    def cos_cov(self, k):
        """Covariance with a cosine signal.

        freq: freq of the cosine signal in Hz

        returns: float covariance
        """
        n = len(self.ys)
        factor = math.pi * k / n
        ys = [math.cos(factor * (i+0.5)) for i in range(n)]
        total = 2 * sum(self.ys * ys)
        return total

    def cos_transform(self):
        """Discrete cosine transform.

        returns: list of frequency, cov pairs
        """
        n = len(self.ys)
        res = []
        for k in range(n):
            cov = self.cos_cov(k)
            res.append((k, cov))

        return res

    def write(self, filename='sound.wav'):
        """Write a wave file.

        filename: string
        """
        print('Writing', filename)
        wfile = WavFileWriter(filename, self.framerate)
        wfile.write(self)
        wfile.close()

    def play(self, filename='sound.wav'):
        """Plays a wave file.
```

```
        filename: string
        """
        self.write(filename)
        play_wave(filename)

    def make_audio(self):
        """Makes an IPython Audio object.
        """
        audio = Audio(data=self.ys.real, rate=self.framerate)
        return audio


def unbias(ys):
    """Shifts a wave array so it has mean 0.

    ys: wave array

    returns: wave array
    """
    return ys - ys.mean()


def 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
    """
    high, low = abs(max(ys)), abs(min(ys))
    return amp * ys / max(high, low)


def shift_right(ys, shift):
    """Shifts a wave array to the right and zero pads.

    ys: wave array
    shift: integer shift

    returns: wave array
    """
    res = np.zeros(len(ys) + shift)
    res[shift:] = ys
    return res


def shift_left(ys, shift):
```

```
    """Shifts a wave array to the left.

    ys: wave array
    shift: integer shift

    returns: wave array
    """
    return ys[shift:]


def truncate(ys, n):
    """Trims a wave array to the given length.

    ys: wave array
    n: integer length

    returns: wave array
    """
    return ys[:n]


def 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
    """
    if max(ys) > 1 or min(ys) < -1:
        warnings.warn('Warning: normalizing before
quantizing.')
        ys = normalize(ys)

    zs = (ys * bound).astype(dtype)
    return zs


def 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
    """
    # a fixed fraction of the segment
    n = len(ys)
    k1 = n // denom

    # a fixed duration of time
    k2 = int(duration * framerate)

    k = min(k1, k2)

    w1 = np.linspace(0, 1, k)
    w2 = np.ones(n - 2*k)
    w3 = np.linspace(1, 0, k)

    window = np.concatenate((w1, w2, w3))
    return ys * window


class Signal:
    """Represents a time-varying signal."""

    def __add__(self, other):
        """Adds two signals.

        other: Signal

        returns: Signal
        """
        if other == 0:
            return self
        return SumSignal(self, other)

    __radd__ = __add__

    @property
    def period(self):
        """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
        """
        return 0.1

    def plot(self, framerate=11025):
```

```
        """Plots the signal.

        The default behavior is to plot three periods.

        framerate: samples per second
        """
        duration = self.period * 3
        wave = self.make_wave(duration, start=0,
framerate=framerate)
        wave.plot()

    def 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
        """
        n = round(duration * framerate)
        ts = start + np.arange(n) / framerate
        ys = self.evaluate(ts)
        return Wave(ys, ts, framerate=framerate)


def 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
    """
    #TODO: confirm that this is never used and remove it
    dt = ts[1] - ts[0]
    framerate = 1.0 / dt
    return framerate


class SumSignal(Signal):
    """Represents the sum of signals."""

    def __init__(self, *args):
        """Initializes the sum.

        args: tuple of signals
        """
        self.signals = args
```

```
    @property
    def period(self):
        """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
        """
        return max(sig.period for sig in self.signals)

    def evaluate(self, ts):
        """Evaluates the signal at the given times.

        ts: float array of times

        returns: float wave array
        """
        ts = np.asarray(ts)
        return sum(sig.evaluate(ts) for sig in self.signals)


class Sinusoid(Signal):
    """Represents a sinusoidal signal."""

    def __init__(self, freq=440, amp=1.0, offset=0,
func=np.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
        """
        self.freq = freq
        self.amp = amp
        self.offset = offset
        self.func = func

    @property
    def period(self):
        """Period of the signal in seconds.

        returns: float seconds
        """
        return 1.0 / self.freq
```

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

        ts: float array of times

        returns: float wave array
        """
        ts = np.asarray(ts)
        phases = PI2 * self.freq * ts + self.offset
        ys = self.amp * self.func(phases)
        return ys


def 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
    """
    return Sinusoid(freq, amp, offset, func=np.cos)


def 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
    """
    return Sinusoid(freq, amp, offset, func=np.sin)


def 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
    """
    return Sinusoid(freq, amp, offset, func=np.sinc)
```

```
class ComplexSinusoid(Sinusoid):
    """Represents a complex exponential signal."""

    def evaluate(self, ts):
        """Evaluates the signal at the given times.

        ts: float array of times

        returns: float wave array
        """
        ts = np.asarray(ts)
        phases = PI2 * self.freq * ts + self.offset
        ys = self.amp * np.exp(1j * phases)
        return ys


class SquareSignal(Sinusoid):
    """Represents a square signal."""

    def evaluate(self, ts):
        """Evaluates the signal at the given times.

        ts: float array of times

        returns: float wave array
        """
        ts = np.asarray(ts)
        cycles = self.freq * ts + self.offset / PI2
        frac, _ = np.modf(cycles)
        ys = self.amp * np.sign(unbias(frac))
        return ys


class SawtoothSignal(Sinusoid):
    """Represents a sawtooth signal."""

    def evaluate(self, ts):
        """Evaluates the signal at the given times.

        ts: float array of times

        returns: float wave array
        """
        ts = np.asarray(ts)
        cycles = self.freq * ts + self.offset / PI2
        frac, _ = np.modf(cycles)
        ys = normalize(unbias(frac), self.amp)
        return ys
```

```
class ParabolicSignal(Sinusoid):
    """Represents a parabolic signal."""

    def evaluate(self, ts):
        """Evaluates the signal at the given times.

        ts: float array of times

        returns: float wave array
        """
        ts = np.asarray(ts)
        cycles = self.freq * ts + self.offset / PI2
        frac, _ = np.modf(cycles)
        ys = (frac - 0.5)**2
        ys = normalize(unbias(ys), self.amp)
        return ys


class CubicSignal(ParabolicSignal):
    """Represents a cubic signal."""

    def evaluate(self, ts):
        """Evaluates the signal at the given times.

        ts: float array of times

        returns: float wave array
        """
        ys = ParabolicSignal.evaluate(self, ts)
        ys = np.cumsum(ys)
        ys = normalize(unbias(ys), self.amp)
        return ys


class GlottalSignal(Sinusoid):
    """Represents a periodic signal that resembles a glottal
signal."""

    def evaluate(self, ts):
        """Evaluates the signal at the given times.

        ts: float array of times

        returns: float wave array
        """
        ts = np.asarray(ts)
        cycles = self.freq * ts + self.offset / PI2
        frac, _ = np.modf(cycles)
        ys = frac**2 * (1-frac)
        ys = normalize(unbias(ys), self.amp)
```

```
        return ys


class TriangleSignal(Sinusoid):
    """Represents a triangle signal."""

    def evaluate(self, ts):
        """Evaluates the signal at the given times.

        ts: float array of times

        returns: float wave array
        """
        ts = np.asarray(ts)
        cycles = self.freq * ts + self.offset / PI2
        frac, _ = np.modf(cycles)
        ys = np.abs(frac - 0.5)
        ys = normalize(unbias(ys), self.amp)
        return ys


class Chirp(Signal):
    """Represents a signal with variable frequency."""

    def __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
        """
        self.start = start
        self.end = end
        self.amp = amp

    @property
    def period(self):
        """Period of the signal in seconds.

        returns: float seconds
        """
        return ValueError('Non-periodic signal.')

    def evaluate(self, ts):
        """Evaluates the signal at the given times.

        ts: float array of times

        returns: float wave array
        """
```

```
        freqs = np.linspace(self.start, self.end, len(ts)-1)
        return self._evaluate(ts, freqs)

    def _evaluate(self, ts, freqs):
        """Helper function that evaluates the signal.

        ts: float array of times
        freqs: float array of frequencies during each interval
        """
        dts = np.diff(ts)
        dps = PI2 * freqs * dts
        phases = np.cumsum(dps)
        phases = np.insert(phases, 0, 0)
        ys = self.amp * np.cos(phases)
        return ys


class ExpoChirp(Chirp):
    """Represents a signal with varying frequency."""

    def evaluate(self, ts):
        """Evaluates the signal at the given times.

        ts: float array of times

        returns: float wave array
        """
        start, end = np.log10(self.start), np.log10(self.end)
        freqs = np.logspace(start, end, len(ts)-1)
        return self._evaluate(ts, freqs)


class SilentSignal(Signal):
    """Represents silence."""

    def evaluate(self, ts):
        """Evaluates the signal at the given times.

        ts: float array of times

        returns: float wave array
        """
        return np.zeros(len(ts))


class Impulses(Signal):
    """Represents silence."""

    def __init__(self, locations, amps=1):
        self.locations = np.asanyarray(locations)
```

```
        self.amps = amps

    def evaluate(self, ts):
        """Evaluates the signal at the given times.

        ts: float array of times

        returns: float wave array
        """
        ys = np.zeros(len(ts))
        indices = np.searchsorted(ts, self.locations)
        ys[indices] = self.amps
        return ys


class _Noise(Signal):
    """Represents a noise signal (abstract parent class)."""

    def __init__(self, amp=1.0):
        """Initializes a white noise signal.

        amp: float amplitude, 1.0 is nominal max
        """
        self.amp = amp

    @property
    def period(self):
        """Period of the signal in seconds.

        returns: float seconds
        """
        return ValueError('Non-periodic signal.')


class UncorrelatedUniformNoise(_Noise):
    """Represents uncorrelated uniform noise."""

    def evaluate(self, ts):
        """Evaluates the signal at the given times.

        ts: float array of times

        returns: float wave array
        """
        ys = np.random.uniform(-self.amp, self.amp, len(ts))
        return ys


class UncorrelatedGaussianNoise(_Noise):
    """Represents uncorrelated gaussian noise."""
```

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

        ts: float array of times

        returns: float wave array
        """
        ys = np.random.normal(0, self.amp, len(ts))
        return ys


class BrownianNoise(_Noise):
    """Represents Brownian noise, aka red noise."""

    def 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
        """
        dys = np.random.uniform(-1, 1, len(ts))
        #ys = scipy.integrate.cumtrapz(dys, ts)
        ys = np.cumsum(dys)
        ys = normalize(unbias(ys), self.amp)
        return ys


class PinkNoise(_Noise):
    """Represents Brownian noise, aka red noise."""

    def __init__(self, amp=1.0, beta=1.0):
        """Initializes a pink noise signal.

        amp: float amplitude, 1.0 is nominal max
        """
        self.amp = amp
        self.beta = beta

    def 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
        """
        signal = UncorrelatedUniformNoise()
        wave = signal.make_wave(duration, start, framerate)
        spectrum = wave.make_spectrum()

        spectrum.pink_filter(beta=self.beta)

        wave2 = spectrum.make_wave()
        wave2.unbias()
        wave2.normalize(self.amp)
        return wave2


def rest(duration):
    """Makes a rest of the given duration.

    duration: float seconds

    returns: Wave
    """
    signal = SilentSignal()
    wave = signal.make_wave(duration)
    return wave


def make_note(midi_num, duration, sig_cons=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
    """
    freq = midi_to_freq(midi_num)
    signal = sig_cons(freq)
    wave = signal.make_wave(duration, framerate=framerate)
    wave.apodize()
    return wave


def make_chord(midi_nums, duration, sig_cons=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
    """
    freqs = [midi_to_freq(num) for num in midi_nums]
    signal = sum(sig_cons(freq) for freq in freqs)
    wave = signal.make_wave(duration, framerate=framerate)
    wave.apodize()
    return wave


def midi_to_freq(midi_num):
    """Converts MIDI note number to frequency.

    midi_num: int MIDI note number

    returns: float frequency in Hz
    """
    x = (midi_num - 69) / 12.0
    freq = 440.0 * 2**x
    return freq


def 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
    """
    signal = SinSignal(freq, offset=offset)
    wave = signal.make_wave(duration)
    return wave


def 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
    """
    signal = CosSignal(freq, offset=offset)
    wave = signal.make_wave(duration)
```

```
    return wave


def mag(a):
    """Computes the magnitude of a numpy array.

    a: numpy array

    returns: float
    """
    return np.sqrt(np.dot(a, a))


def zero_pad(array, n):
    """Extends an array with zeros.

    array: numpy array
    n: length of result

    returns: new NumPy array
    """
    res = np.zeros(n)
    res[:len(array)] = array
    return res


def main():

    cos_basis = cos_wave(440)
    sin_basis = sin_wave(440)

    wave = cos_wave(440, offset=math.pi/2)
    cos_cov = cos_basis.cov(wave)
    sin_cov = sin_basis.cov(wave)
    print(cos_cov, sin_cov, mag((cos_cov, sin_cov)))
    return

    wfile = WavFileWriter()
    for sig_cons in [SinSignal, TriangleSignal,
SawtoothSignal,
                     GlottalSignal, ParabolicSignal,
SquareSignal]:
        print(sig_cons)
        sig = sig_cons(440)
        wave = sig.make_wave(1)
        wave.apodize()
        wfile.write(wave)
    wfile.close()
    return
```

```
    signal = GlottalSignal(440)
    signal.plot()
    pyplot.show()
    return

    wfile = WavFileWriter()
    for m in range(60, 0, -1):
        wfile.write(make_note(m, 0.25))
    wfile.close()
    return

    wave1 = make_note(69, 1)
    wave2 = make_chord([69, 72, 76], 1)
    wave = wave1 | wave2

    wfile = WavFileWriter()
    wfile.write(wave)
    wfile.close()
    return

    sig1 = CosSignal(freq=440)
    sig2 = CosSignal(freq=523.25)
    sig3 = CosSignal(freq=660)
    sig4 = CosSignal(freq=880)
    sig5 = CosSignal(freq=987)
    sig = sig1 + sig2 + sig3 + sig4

    #wave = Wave(sig, duration=0.02)
    #wave.plot()

    wave = sig.make_wave(duration=1)
    #wave.normalize()

    wfile = WavFileWriter(wave)
    wfile.write()
    wfile.close()


if __name__ == '__main__':
    main()
--- === ---

"""This file contains code for use with "Think Stats",
by Allen B. Downey, available from greenteapress.com

Copyright 2014 Allen B. Downey
License: GNU GPLv3 http://www.gnu.org/licenses/gpl.html
"""

from __future__ import print_function
```

```
import math
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

import warnings

# customize some matplotlib attributes
#matplotlib.rc('figure', figsize=(4, 3))

#matplotlib.rc('font', size=14.0)
#matplotlib.rc('axes', labelsize=22.0, titlesize=22.0)
#matplotlib.rc('legend', fontsize=20.0)

#matplotlib.rc('xtick.major', size=6.0)
#matplotlib.rc('xtick.minor', size=3.0)

#matplotlib.rc('ytick.major', size=6.0)
#matplotlib.rc('ytick.minor', size=3.0)


class _Brewer(object):
    """Encapsulates a nice sequence of colors.

    Shades of blue that look good in color and can be
distinguished
    in grayscale (up to a point).

    Borrowed from http://colorbrewer2.org/
    """
    color_iter = None

    colors = ['#f7fbff', '#deebf7', '#c6dbef',
              '#9ecae1', '#6baed6', '#4292c6',
              '#2171b5','#08519c','#08306b'][::-1]

    # lists that indicate which colors to use depending on
how many are used
    which_colors = [[],
                    [1],
                    [1, 3],
                    [0, 2, 4],
                    [0, 2, 4, 6],
                    [0, 2, 3, 5, 6],
                    [0, 2, 3, 4, 5, 6],
                    [0, 1, 2, 3, 4, 5, 6],
                    [0, 1, 2, 3, 4, 5, 6, 7],
                    [0, 1, 2, 3, 4, 5, 6, 7, 8],
```

```
                    ]

    current_figure = None

    @classmethod
    def Colors(cls):
        """Returns the list of colors.
        """
        return cls.colors

    @classmethod
    def ColorGenerator(cls, num):
        """Returns an iterator of color strings.

        n: how many colors will be used
        """
        for i in cls.which_colors[num]:
            yield cls.colors[i]
        raise StopIteration('Ran out of colors in _Brewer.')

    @classmethod
    def InitIter(cls, num):
        """Initializes the color iterator with the given
number of colors."""
        cls.color_iter = cls.ColorGenerator(num)
        fig = plt.gcf()
        cls.current_figure = fig

    @classmethod
    def ClearIter(cls):
        """Sets the color iterator to None."""
        cls.color_iter = None
        cls.current_figure = None

    @classmethod
    def GetIter(cls, num):
        """Gets the color iterator."""
        fig = plt.gcf()
        if fig != cls.current_figure:
            cls.InitIter(num)
            cls.current_figure = fig

        if cls.color_iter is None:
            cls.InitIter(num)

        return cls.color_iter


def _UnderrideColor(options):
    """If color is not in the options, chooses a color.
```

```
    """
    if 'color' in options:
        return options

    # get the current color iterator; if there is none, init
one
    color_iter = _Brewer.GetIter(5)

    try:
        options['color'] = next(color_iter)
    except StopIteration:
        # if you run out of colors, initialize the color
iterator
        # and try again
        warnings.warn('Ran out of colors.  Starting over.')
        _Brewer.ClearIter()
        _UnderrideColor(options)

    return options


def PrePlot(num=None, rows=None, cols=None):
    """Takes hints about what's coming.

    num: number of lines that will be plotted
    rows: number of rows of subplots
    cols: number of columns of subplots
    """
    if num:
        _Brewer.InitIter(num)

    if rows is None and cols is None:
        return

    if rows is not None and cols is None:
        cols = 1

    if cols is not None and rows is None:
        rows = 1

    # resize the image, depending on the number of rows and
cols
    size_map = {(1, 1): (8, 6),
                (1, 2): (12, 6),
                (1, 3): (12, 6),
                (1, 4): (12, 5),
                (1, 5): (12, 4),
                (2, 2): (10, 10),
                (2, 3): (16, 10),
                (3, 1): (8, 10),
```

```
                (4, 1): (8, 12),
                }

    if (rows, cols) in size_map:
        fig = plt.gcf()
        fig.set_size_inches(*size_map[rows, cols])

    # create the first subplot
    if rows > 1 or cols > 1:
        ax = plt.subplot(rows, cols, 1)
        global SUBPLOT_ROWS, SUBPLOT_COLS
        SUBPLOT_ROWS = rows
        SUBPLOT_COLS = cols
    else:
        ax = plt.gca()

    return ax


def SubPlot(plot_number, rows=None, cols=None, **options):
    """Configures the number of subplots and changes the
current plot.

    rows: int
    cols: int
    plot_number: int
    options: passed to subplot
    """
    rows = rows or SUBPLOT_ROWS
    cols = cols or SUBPLOT_COLS
    return plt.subplot(rows, cols, plot_number, **options)


def _Underride(d, **options):
    """Add key-value pairs to d only if key is not in d.

    If d is None, create a new dictionary.

    d: dictionary
    options: keyword args to add to d
    """
    if d is None:
        d = {}

    for key, val in options.items():
        d.setdefault(key, val)

    return d
```

```
def Clf():
    """Clears the figure and any hints that have been set."""
    global LOC
    LOC = None
    _Brewer.ClearIter()
    plt.clf()
    fig = plt.gcf()
    fig.set_size_inches(8, 6)


def Figure(**options):
    """Sets options for the current figure."""
    _Underride(options, figsize=(6, 8))
    plt.figure(**options)


def Plot(obj, ys=None, style='', **options):
    """Plots a line.

    Args:
      obj: sequence of x values, or Series, or anything with
Render()
      ys: sequence of y values
      style: style string passed along to plt.plot
      options: keyword args passed to plt.plot
    """
    options = _UnderrideColor(options)
    label = getattr(obj, 'label', '_nolegend_')
    options = _Underride(options, linewidth=3, alpha=0.7,
label=label)

    xs = obj
    if ys is None:
        if hasattr(obj, 'Render'):
            xs, ys = obj.Render()
        if isinstance(obj, pd.Series):
            ys = obj.values
            xs = obj.index

    if ys is None:
        plt.plot(xs, style, **options)
    else:
        plt.plot(xs, ys, style, **options)


def Vlines(xs, y1, y2, **options):
    """Plots a set of vertical lines.

    Args:
      xs: sequence of x values
```

```
      y1: sequence of y values
      y2: sequence of y values
      options: keyword args passed to plt.vlines
    """
    options = _UnderrideColor(options)
    options = _Underride(options, linewidth=1, alpha=0.5)
    plt.vlines(xs, y1, y2, **options)


def Hlines(ys, x1, x2, **options):
    """Plots a set of horizontal lines.

    Args:
      ys: sequence of y values
      x1: sequence of x values
      x2: sequence of x values
      options: keyword args passed to plt.vlines
    """
    options = _UnderrideColor(options)
    options = _Underride(options, linewidth=1, alpha=0.5)
    plt.hlines(ys, x1, x2, **options)


def axvline(x, **options):
    """Plots a vertical line.

    Args:
      x: x location
      options: keyword args passed to plt.axvline
    """
    options = _UnderrideColor(options)
    options = _Underride(options, linewidth=1, alpha=0.5)
    plt.axvline(x, **options)


def axhline(y, **options):
    """Plots a horizontal line.

    Args:
      y: y location
      options: keyword args passed to plt.axhline
    """
    options = _UnderrideColor(options)
    options = _Underride(options, linewidth=1, alpha=0.5)
    plt.axhline(y, **options)


def tight_layout(**options):
    """Adjust subplots to minimize padding and margins.
    """
```

```
    options = _Underride(options,
                         wspace=0.1, hspace=0.1,
                         left=0, right=1,
                         bottom=0, top=1)
    plt.tight_layout()
    plt.subplots_adjust(**options)


def FillBetween(xs, y1, y2=None, where=None, **options):
    """Fills the space between two lines.

    Args:
      xs: sequence of x values
      y1: sequence of y values
      y2: sequence of y values
      where: sequence of boolean
      options: keyword args passed to plt.fill_between
    """
    options = _UnderrideColor(options)
    options = _Underride(options, linewidth=0, alpha=0.5)
    plt.fill_between(xs, y1, y2, where, **options)


def Bar(xs, ys, **options):
    """Plots a line.

    Args:
      xs: sequence of x values
      ys: sequence of y values
      options: keyword args passed to plt.bar
    """
    options = _UnderrideColor(options)
    options = _Underride(options, linewidth=0, alpha=0.6)
    plt.bar(xs, ys, **options)


def Scatter(xs, ys=None, **options):
    """Makes a scatter plot.

    xs: x values
    ys: y values
    options: options passed to plt.scatter
    """
    options = _Underride(options, color='blue', alpha=0.2,
                         s=30, edgecolors='none')

    if ys is None and isinstance(xs, pd.Series):
        ys = xs.values
        xs = xs.index
```

```
    plt.scatter(xs, ys, **options)


def HexBin(xs, ys, **options):
    """Makes a scatter plot.

    xs: x values
    ys: y values
    options: options passed to plt.scatter
    """
    options = _Underride(options, cmap=matplotlib.cm.Blues)
    plt.hexbin(xs, ys, **options)


def Pdf(pdf, **options):
    """Plots a Pdf, Pmf, or Hist as a line.

    Args:
      pdf: Pdf, Pmf, or Hist object
      options: keyword args passed to plt.plot
    """
    low, high = options.pop('low', None), options.pop('high',
None)
    n = options.pop('n', 101)
    xs, ps = pdf.Render(low=low, high=high, n=n)
    options = _Underride(options, label=pdf.label)
    Plot(xs, ps, **options)


def Pdfs(pdfs, **options):
    """Plots a sequence of PDFs.

    Options are passed along for all PDFs.  If you want
different
    options for each pdf, make multiple calls to Pdf.

    Args:
      pdfs: sequence of PDF objects
      options: keyword args passed to plt.plot
    """
    for pdf in pdfs:
        Pdf(pdf, **options)


def Hist(hist, **options):
    """Plots a Pmf or Hist with a bar plot.

    The default width of the bars is based on the minimum
difference
    between values in the Hist.  If that's too small, you can
```

```
override
    it by providing a width keyword argument, in the same
units
    as the values.

    Args:
      hist: Hist or Pmf object
      options: keyword args passed to plt.bar
    """
    # find the minimum distance between adjacent values
    xs, ys = hist.Render()

    # see if the values support arithmetic
    try:
        xs[0] - xs[0]
    except TypeError:
        # if not, replace values with numbers
        labels = [str(x) for x in xs]
        xs = np.arange(len(xs))
        plt.xticks(xs+0.5, labels)

    if 'width' not in options:
        try:
            options['width'] = 0.9 * np.diff(xs).min()
        except TypeError:
            warnings.warn("Hist: Can't compute bar width
automatically."
                                "Check for non-numeric types in
Hist."
                                "Or try providing width option."
                                )

    options = _Underride(options, label=hist.label)
    options = _Underride(options, align='center')
    if options['align'] == 'left':
        options['align'] = 'edge'
    elif options['align'] == 'right':
        options['align'] = 'edge'
        options['width'] *= -1

    Bar(xs, ys, **options)


def Hists(hists, **options):
    """Plots two histograms as interleaved bar plots.

    Options are passed along for all PMFs.  If you want
different
    options for each pmf, make multiple calls to Pmf.
```

```
    Args:
      hists: list of two Hist or Pmf objects
      options: keyword args passed to plt.plot
    """
    for hist in hists:
        Hist(hist, **options)


def Pmf(pmf, **options):
    """Plots a Pmf or Hist as a line.

    Args:
      pmf: Hist or Pmf object
      options: keyword args passed to plt.plot
    """
    xs, ys = pmf.Render()
    low, high = min(xs), max(xs)

    width = options.pop('width', None)
    if width is None:
        try:
            width = np.diff(xs).min()
        except TypeError:
            warnings.warn("Pmf: Can't compute bar width
automatically."
                          "Check for non-numeric types in
Pmf."
                          "Or try providing width option.")
    points = []

    lastx = np.nan
    lasty = 0
    for x, y in zip(xs, ys):
        if (x - lastx) > 1e-5:
            points.append((lastx, 0))
            points.append((x, 0))

        points.append((x, lasty))
        points.append((x, y))
        points.append((x+width, y))

        lastx = x + width
        lasty = y
    points.append((lastx, 0))
    pxs, pys = zip(*points)

    align = options.pop('align', 'center')
    if align == 'center':
        pxs = np.array(pxs) - width/2.0
    if align == 'right':
```

```
        pxs = np.array(pxs) - width

    options = _Underride(options, label=pmf.label)
    Plot(pxs, pys, **options)


def Pmfs(pmfs, **options):
    """Plots a sequence of PMFs.

    Options are passed along for all PMFs.  If you want
different
    options for each pmf, make multiple calls to Pmf.

    Args:
      pmfs: sequence of PMF objects
      options: keyword args passed to plt.plot
    """
    for pmf in pmfs:
        Pmf(pmf, **options)


def Diff(t):
    """Compute the differences between adjacent elements in a
sequence.

    Args:
        t: sequence of number

    Returns:
        sequence of differences (length one less than t)
    """
    diffs = [t[i+1] - t[i] for i in range(len(t)-1)]
    return diffs


def Cdf(cdf, complement=False, transform=None, **options):
    """Plots a CDF as a line.

    Args:
      cdf: Cdf object
      complement: boolean, whether to plot the complementary
CDF
      transform: string, one of 'exponential', 'pareto',
'weibull', 'gumbel'
      options: keyword args passed to plt.plot

    Returns:
      dictionary with the scale options that should be passed
to
      Config, Show or Save.
```

```
    """
    xs, ps = cdf.Render()
    xs = np.asarray(xs)
    ps = np.asarray(ps)

    scale = dict(xscale='linear', yscale='linear')

    for s in ['xscale', 'yscale']:
        if s in options:
            scale[s] = options.pop(s)

    if transform == 'exponential':
        complement = True
        scale['yscale'] = 'log'

    if transform == 'pareto':
        complement = True
        scale['yscale'] = 'log'
        scale['xscale'] = 'log'

    if complement:
        ps = [1.0-p for p in ps]

    if transform == 'weibull':
        xs = np.delete(xs, -1)
        ps = np.delete(ps, -1)
        ps = [-math.log(1.0-p) for p in ps]
        scale['xscale'] = 'log'
        scale['yscale'] = 'log'

    if transform == 'gumbel':
        xs = np.delete(xs, 0)
        ps = np.delete(ps, 0)
        ps = [-math.log(p) for p in ps]
        scale['yscale'] = 'log'

    options = _Underride(options, label=cdf.label)
    Plot(xs, ps, **options)
    return scale


def Cdfs(cdfs, complement=False, transform=None, **options):
    """Plots a sequence of CDFs.

    cdfs: sequence of CDF objects
    complement: boolean, whether to plot the complementary CDF
    transform: string, one of 'exponential', 'pareto',
'weibull', 'gumbel'
    options: keyword args passed to plt.plot
    """
```

```
    for cdf in cdfs:
        Cdf(cdf, complement, transform, **options)


def Contour(obj, pcolor=False, contour=True, imshow=False,
**options):
    """Makes a contour plot.

    d: map from (x, y) to z, or object that provides GetDict
    pcolor: boolean, whether to make a pseudocolor plot
    contour: boolean, whether to make a contour plot
    imshow: boolean, whether to use plt.imshow
    options: keyword args passed to plt.pcolor and/or
plt.contour
    """
    try:
        d = obj.GetDict()
    except AttributeError:
        d = obj

    _Underride(options, linewidth=3, cmap=matplotlib.cm.Blues)

    xs, ys = zip(*d.keys())
    xs = sorted(set(xs))
    ys = sorted(set(ys))

    X, Y = np.meshgrid(xs, ys)
    func = lambda x, y: d.get((x, y), 0)
    func = np.vectorize(func)
    Z = func(X, Y)

    x_formatter =
matplotlib.ticker.ScalarFormatter(useOffset=False)
    axes = plt.gca()
    axes.xaxis.set_major_formatter(x_formatter)

    if pcolor:
        plt.pcolormesh(X, Y, Z, **options)
    if contour:
        cs = plt.contour(X, Y, Z, **options)
        plt.clabel(cs, inline=1, fontsize=10)
    if imshow:
        extent = xs[0], xs[-1], ys[0], ys[-1]
        plt.imshow(Z, extent=extent, **options)


def Pcolor(xs, ys, zs, pcolor=True, contour=False, **options):
    """Makes a pseudocolor plot.

    xs:
```

```
    ys:
    zs:
    pcolor: boolean, whether to make a pseudocolor plot
    contour: boolean, whether to make a contour plot
    options: keyword args passed to plt.pcolor and/or
plt.contour
    """
    _Underride(options, linewidth=3, cmap=matplotlib.cm.Blues)

    X, Y = np.meshgrid(xs, ys)
    Z = zs

    x_formatter =
matplotlib.ticker.ScalarFormatter(useOffset=False)
    axes = plt.gca()
    axes.xaxis.set_major_formatter(x_formatter)

    if pcolor:
        plt.pcolormesh(X, Y, Z, **options)

    if contour:
        cs = plt.contour(X, Y, Z, **options)
        plt.clabel(cs, inline=1, fontsize=10)


def Text(x, y, s, **options):
    """Puts text in a figure.

    x: number
    y: number
    s: string
    options: keyword args passed to plt.text
    """
    options = _Underride(options,
                         fontsize=16,
                         verticalalignment='top',
                         horizontalalignment='left')
    plt.text(x, y, s, **options)


LEGEND = True
LOC = None

def Config(**options):
    """Configures the plot.

    Pulls options out of the option dictionary and passes
them to
    the corresponding plt functions.
    """
```

```
    names = ['title', 'xlabel', 'ylabel', 'xscale', 'yscale',
             'xticks', 'yticks', 'axis', 'xlim', 'ylim']

    for name in names:
        if name in options:
            getattr(plt, name)(options[name])

    global LEGEND
    LEGEND = options.get('legend', LEGEND)

    # see if there are any elements with labels;
    # if not, don't draw a legend
    ax = plt.gca()
    handles, labels = ax.get_legend_handles_labels()

    if LEGEND and len(labels) > 0:
        global LOC
        LOC = options.get('loc', LOC)
        frameon = options.get('frameon', True)

        try:
            plt.legend(loc=LOC, frameon=frameon)
        except UserWarning:
            pass

    # x and y ticklabels can be made invisible
    val = options.get('xticklabels', None)
    if val is not None:
        if val == 'invisible':
            ax = plt.gca()
            labels = ax.get_xticklabels()
            plt.setp(labels, visible=False)

    val = options.get('yticklabels', None)
    if val is not None:
        if val == 'invisible':
            ax = plt.gca()
            labels = ax.get_yticklabels()
            plt.setp(labels, visible=False)

def set_font_size(title_size=16, label_size=16,
ticklabel_size=14, legend_size=14):
    """Set font sizes for the title, labels, ticklabels, and
legend.
    """
    def set_text_size(texts, size):
        for text in texts:
            text.set_size(size)

    ax = plt.gca()
```

```
    # TODO: Make this function more robust if any of these
elements
    # is missing.

    # title
    ax.title.set_size(title_size)

    # x axis
    ax.xaxis.label.set_size(label_size)
    set_text_size(ax.xaxis.get_ticklabels(), ticklabel_size)

    # y axis
    ax.yaxis.label.set_size(label_size)
    set_text_size(ax.yaxis.get_ticklabels(), ticklabel_size)

    # legend
    legend = ax.get_legend()
    if legend is not None:
        set_text_size(legend.texts, legend_size)


def bigger_text():
    sizes = dict(title_size=16, label_size=16,
ticklabel_size=14, legend_size=14)
    set_font_size(**sizes)


def Show(**options):
    """Shows the plot.

    For options, see Config.

    options: keyword args used to invoke various plt functions
    """
    clf = options.pop('clf', True)
    Config(**options)
    plt.show()
    if clf:
        Clf()


def Plotly(**options):
    """Shows the plot.

    For options, see Config.

    options: keyword args used to invoke various plt functions
    """
    clf = options.pop('clf', True)
```

```
    Config(**options)
    import plotly.plotly as plotly
    url = plotly.plot_mpl(plt.gcf())
    if clf:
        Clf()
    return url


def Save(root=None, formats=None, **options):
    """Saves the plot in the given formats and clears the
figure.

    For options, see Config.

    Note: With a capital S, this is the original save,
maintained for
    compatibility.  New code should use save(), which works
better
    with my newer code, especially in Jupyter notebooks.

    Args:
      root: string filename root
      formats: list of string formats
      options: keyword args used to invoke various plt
functions
    """
    clf = options.pop('clf', True)

    save_options = {}
    for option in ['bbox_inches', 'pad_inches']:
        if option in options:
            save_options[option] = options.pop(option)

    # TODO: falling Config inside Save was probably a
mistake, but removing
    # it will require some work
    Config(**options)

    if formats is None:
        formats = ['pdf', 'png']

    try:
        formats.remove('plotly')
        Plotly(clf=False)
    except ValueError:
        pass

    if root:
        for fmt in formats:
            SaveFormat(root, fmt, **save_options)
```

```
    if clf:
        Clf()


def save(root, formats=None, **options):
    """Saves the plot in the given formats and clears the
figure.

    For options, see plt.savefig.

    Args:
      root: string filename root
      formats: list of string formats
      options: keyword args passed to plt.savefig
    """
    if formats is None:
        formats = ['pdf', 'png']

    try:
        formats.remove('plotly')
        Plotly(clf=False)
    except ValueError:
        pass

    for fmt in formats:
        SaveFormat(root, fmt, **options)


def SaveFormat(root, fmt='eps', **options):
    """Writes the current figure to a file in the given
format.

    Args:
      root: string filename root
      fmt: string format
    """
    _Underride(options, dpi=300)
    filename = '%s.%s' % (root, fmt)
    print('Writing', filename)
    plt.savefig(filename, format=fmt, **options)


# provide aliases for calling functions with lower-case names
preplot = PrePlot
subplot = SubPlot
clf = Clf
figure = Figure
plot = Plot
vlines = Vlines
hlines = Hlines
```

```
fill_between = FillBetween
text = Text
scatter = Scatter
pmf = Pmf
pmfs = Pmfs
hist = Hist
hists = Hists
diff = Diff
cdf = Cdf
cdfs = Cdfs
contour = Contour
pcolor = Pcolor
config = Config
show = Show


def main():
    color_iter = _Brewer.ColorGenerator(7)
    for color in color_iter:
        print(color)


if __name__ == '__main__':
    main()
```