```
signal = thinkdsp.SawtoothSignal(freq=500)
wave = signal.make_wave(duration=0.1, framerate=10000)
hs = dft(wave.ys)
amps = np.absolute(hs)
```

This code makes a sawtooth wave with frequency 500 Hz, sampled at framerate 10 kHz. `hs` contains the complex DFT of the wave; `amps` contains the amplitude at each frequency. But what frequency do these amplitudes correspond to? If we look at the body of `dft`, we see:

```
fs = np.arange(N)
```

It's tempting to think that these values are the right frequencies. The problem is that `dft` doesn't know the sampling rate. The DFT assumes that the duration of the wave is 1 time unit, so it thinks the sampling rate is N per time unit. In order to interpret the frequencies, we have to convert from these arbitrary time units back to seconds, like this:

```
fs = np.arange(N) * framerate / N
```

With this change, the range of frequencies is from 0 to the actual framerate, 10 kHz. Now we can plot the spectrum:

```
plt.plot(fs, amps)
```

Figure 7.3 shows the amplitude of the signal for each frequency component from 0 to 10 kHz. The left half of the figure is what we should expect: the dominant frequency is at 500 Hz, with harmonics dropping off like $1/f$.

But the right half of the figure is a surprise. Past 5000 Hz, the amplitude of the harmonics start growing again, peaking at 9500 Hz. What's going on?

The answer: aliasing. Remember that with framerate 10000 Hz, the folding frequency is 5000 Hz. As we saw in Section 2.3, a component at 5500 Hz is indistinguishable from a component at 4500 Hz. When we evaluate the DFT at 5500 Hz, we get the same value as at 4500 Hz. Similarly, the value at 6000 Hz is the same as the one at 4000 Hz, and so on.

The DFT of a real signal is symmetric around the folding frequency. Since there is no additional information past this point, we can save time by evaluating only the first half of the DFT, and that's exactly what `np.fft.rfft` does.

7.10 Exercises

Solutions to these exercises are in `chap07soln.ipynb`.

Exercise 7.1 The notebook for this chapter, `chap07.ipynb`, contains additional examples and explanations. Read through it and run the code.

Exercise 7.2 In this chapter, I showed how we can express the DFT and inverse DFT as matrix multiplications. These operations take time proportional to N^2, where N is the length of the wave array. That is fast enough for many applications, but there is a faster algorithm, the Fast Fourier Transform (FFT), which takes time proportional to $N \log N$.

The key to the FFT is the Danielson-Lanczos lemma:

$$DFT(y)[n] = DFT(e)[n] + exp(-2\pi in/N)DFT(o)[n]$$

Where $DFT(y)[n]$ is the nth element of the DFT of y; e is a wave array containing the even elements of y, and o contains the odd elements of y.

This lemma suggests a recursive algorithm for the DFT:

1. Given a wave array, y, split it into its even elements, e, and its odd elements, o.
2. Compute the DFT of e and o by making recursive calls.
3. Compute $DFT(y)$ for each value of n using the Danielson-Lanczos lemma.

For the base case of this recursion, you could wait until the length of y is 1. In that case, $DFT(y) = y$. Or if the length of y is sufficiently small, you could compute its DFT by matrix multiplication, possibly using a precomputed matrix.

Hint: I suggest you implement this algorithm incrementally by starting with a version that is not truly recursive. In Step 2, instead of making a recursive call, use `dft`, as defined by Section 7.7, or `np.fft.fft`. Get Step 3 working, and confirm that the results are consistent with the other implementations. Then add a base case and confirm that it works. Finally, replace Step 2 with recursive calls.

One more hint: Remember that the DFT is periodic; you might find `np.tile` useful.

You can read more about the FFT at `https://en.wikipedia.org/wiki/Fast_Fourier_transform`.