Classes and Objects

Interfaces
Classes and objects help you separate *interface* from *implementation*.
Classes and objects help you separate *interface* from *implementation*.

Interface: what something knows how to do.
Classes and objects help you separate *interface* from *implementation*

Interface: what something knows how to do
Implementation: how it does things
Classes and objects help you separate *interface* from *implementation*.

**Interface**: what something knows how to do

**Implementation**: how it does things

Programming to interfaces makes it (much) easier to test/change/replace parts of a program.
Classes and objects help you separate interface from implementation.

Interface: what something knows how to do
Implementation: how it does things

Programming to interfaces makes it (much) easier to test/change/replace parts of a program.

Explain by example.
Starting point: irregular time series signal
Starting point: irregular time series signal

Hide irregularity by allowing sampling at any time
Starting point: irregular time series signal

Hide irregularity by allowing sampling at any time

Step function
Starting point: irregular time series signal

Hide irregularity by allowing sampling at any time

- Step function
- Linear interpolation
Define the interface first
Define the interface first

class SomeClassName(object):

    def __init__(self, values):
        '''Values is ((x0, y0), (x1, y1), ...)'''
        store values
Define the interface first

class SomeClassName(object):

    def __init__(self, values):
        '''Values is ((x0, y0), (x1, y1), ...)'''
        store values

    def get(self, where):
        if where is out of bounds:
            raise exception
        else:
            return interpolated value
First implementation

class StepSignal(object):

    def __init__(self, values):
        self.values = values[:]  # make a copy
First implementation

class StepSignal(object):
    ...
    def get(self, where):
        if where < self.values[0][0]:
            raise IndexError, '%f too low' % where
        for i in range(len(self.values)-1):
            x0, y0 = self.values[i]
            x1, y1 = self.values[i+1]
            if x0 <= where <= x1:
                return y0
        raise IndexError, '%f too high' % where
Test a few points

```python
interp = StepSignal(((0., 0.), (1., 1.), (2., 2.)))

for x in (0.0, 0.5, 1.0, 1.75):
    print x, interp.get(x)

0.0 0.0
0.5 0.0
1.0 1.0
1.75 1.0
```
Test error handling too

```python
for val in (-100.0, -0.0001, 2.0, 100.0):
    try:
        interp.get(val)
        assert False, 'Should not be here:', val
    except IndexError, e:
        print val, 'raised expected exception'
-100.0 raised expected exception
-0.0001 raised expected exception
2.0 raised expected exception
100.0 raised expected exception
```
Now create second implementation

class LinearSignal(object):
    ...
    def get(self, where):
        if where < self.values[0][0]:
            raise IndexError, '%f too low' % where
        for i in range(len(self.values)-1):
            x0, y0 = self.values[i]
            x1, y1 = self.values[i+1]
            if x0 <= where <= x1:
                return y0 + (y1-y0) * (where-x0) / (x1-x0)
        raise IndexError, '%f too high' % where
Now create second implementation

class LinearSignal(object):
    ...

    def get(self, where):
        if where < self.values[0][0]:
            raise IndexError, '%f too low' % where
        for i in range(len(self.values)-1):
            x0, y0 = self.values[i]
            x1, y1 = self.values[i+1]
            if x0 <= where <= x1:
                return y0 + (y1-y0) * (where-x0) / (x1-x0)
        raise IndexError, '%f too high' % where
Test it as well

```python
interp = LinearSignal(((0., 0.), (1., 1.), (2., 2.)))
for x in (0.0, 0.5, 1.0, 1.75):
    print x, interp.get(x)
0.0 0.0
0.5 0.5
1.0 1.0
1.75 1.75
```
Test it as well

```python
interp = LinearSignal(((0., 0.), (1., 1.),
                      (2., 2.)))
for x in (0.0, 0.5, 1.0, 1.75):
    print x, interp.get(x)
0.0 0.0
0.5 0.5
1.0 1.0
1.75 1.75
```
Test it as well

```python
interp = LinearSignal(((0., 0.), (1., 1.),
   (2., 2.)))

for x in (0.0, 0.5, 1.0, 1.75):
    print x, interp.get(x)
```

0.0 0.0
0.5 0.5
1.0 1.0
1.75 1.75

Error handling still works
And now the payoff
def average(signal, x0, x1, num_samples):
    width = (x1 - x0) / num_samples
    total = 0.0
    for i in range(num_samples):
        x = x0 + i * width
        total += signal.get(x)
    return total / num_samples
And now the payoff

```python
def average(signal, x0, x1, num_samples):
    width = (x1 - x0) / num_samples
    total = 0.0
    for i in range(num_samples):
        x = x0 + i * width
        total += signal.get(x)
    return total / num_samples
```
And now the payoff

```python
def average(signal, x0, x1, num_samples):
    width = (x1 - x0) / num_samples
    total = 0.0
    for i in range(num_samples):
        x = x0 + i * width
        total += signal.get(x)
    return total / num_samples
```

Can use an object of either class for signal
And now the payoff

```python
def average(signal, x0, x1, num_samples):
    width = (x1 - x0) / num_samples
    total = 0.0
    for i in range(num_samples):
        x = x0 + i * width
        total += signal.get(x)
    return total / num_samples
```

Can use an object of either class for signal

Or an object of a class that doesn't exist yet
For example

class Sinusoid(object):

    def __init__(self, amplitude, frequency):
        self.amp = amplitude
        self.freq = frequency

    def get(self, x):
        return self.amp * math.sin(x * self.freq)
For example:

```python
class Sinusoid(object):

    def __init__(self, amplitude, frequency):
        self.amp = amplitude
        self.freq = frequency

    def get(self, x):
        return self.amp * math.sin(x * self.freq)
```

Clear interfaces make code more extensible
For exampleé

```python
class Sinusoid(object):
    def __init__(self, amplitude, frequency):
        self.amp = amplitude
        self.freq = frequency

    def get(self, x):
        return self.amp * math.sin(x * self.freq)
```

Clear interfaces make code more extensible

Only care about actual class when constructing
created by

Greg Wilson

January 2011