Classes and Objects

Overloading

A *sparse vector* is one whose values are mostly zero

Instead of storing those zeroes, store (index, value) pairs in a dictionary for non-zero elements

Build the class

Then see how to make it look like a built-in class

```
class SparseVector(object):

  def __init__(self, len):
    self.values = {}
    self.len = len

  def get(self, index):
    assert 0 <= index < self.len, 'Index out
   of range'
    return self.get(index, 0.0)

  def set(self, index, value):
    assert 0 <= index < self.len, 'Index out
   of range'
    self.values[index] = value
```

Add a few more methods

But still easy to tell our classes from Python's `list`

Python list	Our vector class
`len(vec)`	`vec.length()`
`vec[i] = 0.0`	`vec.set(i,0.0)`
`if x in vec`	`vec.contains(x)`

Make `SparseVector` look like a list by *overloading* the built-in operators

(Almost) everything in Python is a method call

Built-in functions like `len` look for specific methods

```
class SparseVector(object):
  ...
  def __len__(self):
    return self.len

sv = SparseVector(10)
print len(sv)
10
```

"If the object has a `__len__` method, return its result"

Special syntax like `v[i]` also looks for special methods

```
class SparseVector(object):
  ...
  def __getitem__(self, index):
    return self.values.get(index, 0)

sv = SparseVector(10)
print sv[3]
0.0
```

"If the object has a `__getitem__` method, return its result"

Not quite right

```
>>> alpha = [1.0, 0.0, 3.0]
>>> alpha[5]
IndexError: list index out of range
>>> beta = SparseVector(3)
>>> beta[5]
KeyError: 5
```

Code that was using a list will notice a difference if we give it a `SparseVector` instead

"Works the same way" includes "fails the same way"

Better implementation

```
class SparseVector(object):
  ...
  def __getitem__(self, index):
    self.check_index(index)
    return self.values.get(index, 0)

  def check_index(self, index):
    if (index < 0) or (index >= self.len):
      raise IndexError('index out of range')
```

If we can get, we should be able to set

```
class SparseVector(object):
  ...
  def __setitem__(self, index, value):
    self.check_index(index)
    return self.values[index] = value

>>> sv = SparseVector(10)
>>> sv[5] = 3.0
>>> print sv[0], sv[5], sv[9]
0.0 3.0 0.0
```

Another kind of vector

```
class Vec2d(object):

  def __init__(self, x=0.0, y=0.0):
    self.x, self.y = x, y

  def  __add__(self, other):
    return Vec2d(self.x + other.x, self.y +
  other.y)

  def  __sub__(self, other):
    return Vec2d(self.x - other.x, self.y -
  other.y)
```

Try it out

```
>>> one = Vec2d(1.0, 1.0)
>>> two = Vec2d(2.0, 2.0)
>>> three = one + two
>>> print three.x, three.y
3.0 3.0
```

So far, so good, but:

```
>>> print one
<__main__.Vec2d object at 0x01CF23B0>
```

Tell vectors how to represent themselves as strings

```
class Vec2d(object):
  ...
  def __str__(self):
    return '[%f, %f]' % (self.x, self.y)

>>> one = Vec2d(1.0, 1.0)
>>> str(one)
[1.0, 1.0]
>>> print one
[1.0, 1.0]
```

"If the object has a `__str__` method,
call it"

Why create new objects?

```
class Vec2d(object):
  ...
  def __add__(self, other):
    return Vec2d(self.x + other.x, self.y +
  other.y)
```

Because `x + y` doesn't modify either `x` or `y`

Define `__iadd__` for in-place addition

Can mix types

```
class Vec2d(object):
  ...
  def __mul__(self, scalar):
    return Vec2d(self.x * scalar, self.y *
   scalar)

>>> one = Vec2d(1.0, 1.0)
>>> print one * 3
[3.0, 3.0]
>>> print 3 * one
TypeError: unsupported operand type(s) for *:
'int' and 'Vec2d'
```

`x * y` is `x.__mul__(y)` (if `x` has a `__mul__` method)

`int` doesn't

But we can give `Vec2d` an `__rmul__` method

'r' meaning "right hand side"

Because some operators

don't commute

- `x * y` is `x.__mul__(y)` (if `x` has a `__mul__` method)
- `int` doesn't
- But we can give `Vec2d` an `__rmul__` method

```
class Vec2d(object):
  ...
  def __rmul__(self, scalar):
    return Vec2d(self.x * scalar, self.y * scalar)

>>> one = Vec2d(1.0, 1.0)
>>> print 3 * one
[3.0, 3.0]
```

What if we want `Vec2d * Vec2d` to be dot product?

```
class Vec2d(object):
  ...
  def __mul__(self, other):
    if type(other) is Vec2d:
      return dot(self, other)
    else:
      return Vec2d(self.x * other, self.y *
   other)
```

Should check that `other` is a number, not a string

Gets us back to the `if`/`elif`/`elif`/é on types

that objects were invented to avoid

This is (partly) why some languages declare types

```
class Vec2d extends object {
  ...
  float __mul__(Vec2d other) {
    return this.x * other.x + this.y *
   other.y;
  }
  Vec2d __mul__(float other) {
    return new Vec2d(this.x * other, this.y *
   other);
  }
}
```

`Vec2d * string` won't even compile (no such method)

Makes simple things harder, but hard things easier

Either way, the goal is *polymorphism*

```
def dot(left, right):
  assert len(left) == len(right), 'Length
   mismatch'
  result = 0.0
  for i in range(len(left)):
    result += left[i] * right[i]
  return result

>>> sv = SparseVector(3)
>>> sv[2] = 9.0
>>> print dot(sv, [1.0, 2.0, 3.0])
27
```

Or list with list, or sparse vector with sparse vector, or é

With great power comes great responsibility

In C++, `x << y` means either:

ï shift the bits in `x` to the left by `y` places, or

ï print `y` to the open file `x`

Only overload when:

ï there is a strong analogy to an existing type

ï it is possible to reproduce *exactly* that type's behavior (including error behavior)

software carpentry

created by

Greg Wilson

December 2010