Jonas van den Brink, jvbrink

j.v.d.brink@fys.uio.no

7.1— Extend the class from Exercise 6.5

```
from vec3d import Vec3D as Vec3D_base

class Vec3D(Vec3D_base):
    '''Extension of the Vec3D class created in week 6'''

    def __add__(self, other):
        '''Add should now also handle scalars'''
        if isinstance(other, Vec3D):
            # Normal vector addition
            return Vec3D_base.__add__(self, other)

        elif isinstance(other, (int, float)):
            # Add the scalar to each component
            x, y, z = self.coordinates
            return Vec3D(x+other, y+other, z+other)

    def __radd__(self, other):
        '''Addition is commutative, so the reverse addition
        can simply call the Vec3D addition method.'''
        return self.__add__(other)

    def __sub__(self, other):
        '''Sub should now also handle scalars'''
        if isinstance(other, Vec3D):
            # Normal vector subtraction
            return Vec3D_base.__sub__(self, other)
        elif isinstance(other, (int, float)):
            # Subtract the scalar from each component
            x, y, z = self.coordinates
            return Vec3D(x-other, y-other, z-other)

    def __rsub__(self, other):
        '''Subtraction is not commutative, so we define
         the reverse subtraction method explicitly.'''
        x, y, z = self.coordinates
        return (other-x, other-y, other-z)

    def __mul__(self, other):
        '''Should handle multiplication by scalar'''
        if isinstance(other, Vec3D):
            # Normal dot product
            return Vec3D_base.__mul__(self, other)
        elif isinstance(other, (int, float)):
            # Multiply each component by the scalar
            x, y, z = self.coordinates
            return Vec3D(x*other, y*other, z*other)

    def __rmul__(self, other):
        '''Multiplication is commutative'''
        return self.__mul__(other)

    def __div__(self, other):
        if isinstance(other, Vec3D):
            raise TypeError('Vector divided by vector is not defined')
        elif isinstance(other, (int, float)):
            # Divide each component by the scalar
            x, y, z = self.coordinates
            c = float(other)
            return Vec3D(x/c, y/c, z/c)

    def __rdiv__(self, other):
        raise TypeError('Scalar divided by vector is not defined')
```

```python
from vec3d_ext import Vec3D

u = Vec3D(1, 0, 0)
v = Vec3D(0, -0.5, 2)

a = 1.5
b = 1
c = 2.5

print "Test vectors:"
print "u: ", u
print "v: ", v

print "\nTesting addition:"
print "u+v: ", u+v # Normal vector addition
print "a+v: ", a+v # Reverse scalar addition
print "v+a: ", v+a # Forward scalar addition
print "b+u: ", b+u # Test for integer (instead of float)

print "\nTesting subtraction:"
print "u-v: ", u-v # Normal vector subtraction
print "v-a: ", v-a # Forward scalar subtraction
print "a-v: ", a-v # Reverse scalar subtraction

print "\nTesting multiplication"
print "u*v: ", u*v # Normal dot product
print "c*u: ", c*u # Reverse multiplication by scalar
print "u*c: ", u*c # Forward multiplication by scalar

print "\nTesting division by scalar, and error-handling"
print "u/c: ", u/c # Forward division by scalar

print "\nTesting scalar divided by vector"
try:
    print c/u
except TypeError as error:
    print "c/u: TypeError:", error

print "\nTesting vector divided by vector"
try:
    print u/v
except TypeError as error:
    print "u/v: TypeError:", error

'''
user$ python vec3d_ext_example.py
Test vectors:
u:  (1, 0, 0)
v:  (0, -0.5, 2)

Testing addition:
u+v:  (1, -0.5, 2)
a+v:  (1.5, 1, 3.5)
v+a:  (1.5, 1, 3.5)
b+u:  (2, 1, 1)

Testing subtraction:
u-v:  (1, 0.5, -2)
v-a:  (-1.5, -2, 0.5)
a-v:  (1.5, 2.0, -0.5)

Testing multiplication
u*v:  0.0
c*u:  (2.5, 0, 0)
u*c:  (2.5, 0, 0)

Testing division by scalar, and error-handling
u/c:  (0.4, 0, 0)

Testing scalar divided by vector
c/u: TypeError: Scalar divided by vector is not defined

Testing vector divided by vector
u/v: TypeError: Vector divided by vector is not defined
'''
```

7.2—Vectorize a constant function

```
import numpy as np

def initial_condition(x):
    y = np.empty(np.shape(x))
    y.fill(3.0)
    return y

if __name__ == '__main__':
    shape = (5, 3)
    x = np.random.random(shape)
    y = initial_condition(x)
    print y.shape
    print y
    print initial_condition(7)

'''
user$ vectorize_function.py
(5, 3)
[[ 3.  3.  3.]
 [ 3.  3.  3.]
 [ 3.  3.  3.]
 [ 3.  3.  3.]
 [ 3.  3.  3.]]
3.0
'''
```

7.3—Vectorize a numerical integration rule

```python
import numpy as np

def trapez(f, a, b, n):
    '''Integrate f(x) on x in [a,b] using trapezoidal rule
    with a mesh of n cells.'''

    h = (b-a)/float(n)
    s =  h/2.*(f(a) + f(b))
    for i in range(1, n):
        s += h * f(a+i*h)

    return s

def trapez_vectorized(f, a, b, n):
    '''A vectorized version of the trapezoidal function.'''

    x = np.linspace(a, b, n+1)
    h = x[1]-x[0]
    y = h*f(x)
    y[0]  /= 2.
    y[-1]  /= 2.

    return np.sum(y)

if __name__ == '__main__':
    # Test the difference in run-times between the functions
    import numpy as np
    import timeit

    # Integrands to be tested
    f_1 = lambda x: 1+x
    f_2 = lambda x: np.exp(-x*x)*np.log(x + x*np.sin(x))

    # The setup is used by timeit, but does not contribute to the final time
    setup = "from __main__ import f_1, f_2, trapez, trapez_vectorized"

    def time_function(integrator, f, a, b, n):
        # Define the command to be timed
        command = "%s(%s, %g, %g, %d)" % (integrator, f, a, b, n)
        # Run and time the command
        t = timeit.timeit(command, setup=setup, number = 1)
        return t

    for f in ['f_1', 'f_2']:
        print "\nTimes for %s\n  n   %8s%14s" % (f, 'loop', 'vectorized')
        for n in [1e3, 1e4, 1e5, 1e6, 1e7]:
            t1 = time_function('trapez', f, 1, 5, n)
            t2 = time_function('trapez_vectorized', f, 1, 5, n)
            print "%5.0e%10.2e%11.2e"  % (n, t1, t2)

'''
user$ python vectorized_integration.py

Times for f_1
  n       loop    vectorized
1e+03   4.66e-04   1.15e-04
1e+04   3.41e-03   3.02e-04
1e+05   3.69e-02   2.71e-03
1e+06   3.64e-01   4.24e-02
1e+07   3.67e+00   4.32e-01

Times for f_2
  n       loop    vectorized
1e+03   9.99e-03   2.18e-04
1e+04   9.81e-02   1.39e-03
1e+05   9.86e-01   1.27e-02
1e+06   9.99e+00   1.69e-01
1e+07   1.00e+02   2.02e+00
'''
```

7.4—Make a class for sparse vectors

```
class SparseVec:
    '''Class for representing a sparse vector of a specified size'''

    def __init__(self, size):
        self.size = size
        self.elements = {k:0 for k in range(size)}

    def __getitem__(self, key):
        # Need to explicitily check the index
        if key > (self.size - 1):
            raise IndexError('index out of range')
        return self.elements[key]

    def __setitem__(self, key, value):
        # Need to explicitily check the index
        if key > (self.size - 1):
            raise IndexError('index out of range')
        self.elements[key] = value

    def __str__(self):
        s = ['[%d]=%g' % (k, self.elements[k]) for k in self.elements.keys()]
        return ' '.join(s)

    def __len__(self):
        return self.size

    def __add__(self, other):
        # Adds to sparse vectors together, defining a third.
        new = SparseVec(max(self.size, other.size))
        for ai, i in self:
            new[i] += ai
        for ai, i in other:
            new[i] += ai
        return new

    def __iter__(self):
        # Returns iterator
        self.k = 0
        return self

    def nonzeros(self):
        # Returns the non-zero elements as a dictionary
        d = self.elements
        return {k:d[k] for k in d.keys() if d[k]}

    def next(self):
        # Increment iterator object by one, raise stop when done
        k = self.k
        self.k += 1
        if k == self.size:
            raise StopIteration
        else:
            return self.elements[k], k
```

(Example-run is shown on the next page.)

```python
'''
Program for testing the SparseVec class
'''

from SparseVec import SparseVec

a = SparseVec(4)
a[2] = 9.2
a[0] = -1
print "a:\n\t", a
print "a.nonszeros():\n\t", a.nonzeros()

b = SparseVec(5)
b[1] = 1
print "b:\n\t", b
print "b:\n\t", b.nonzeros()

c = a + b
print "c:\n\t", c
print "c.nonzeros():\n\t", c.nonzeros()

print "Testing iterator"
for ai, i in a:  # SparseVec iterator
    print 'a[%d]=%g ' % (i, ai),


'''
user$ python SparseVec_example.py
a:
    [0]=-1 [1]=0 [2]=9.2 [3]=0
a.nonszeros():
    {0: -1, 2: 9.2}
b:
    [0]=0 [1]=1 [2]=0 [3]=0 [4]=0
b:
    {1: 1}
c:
    [0]=-1 [1]=1 [2]=9.2 [3]=0 [4]=0
c.nonzeros():
    {0: -1, 1: 1, 2: 9.2}
'''
```

7.5—Implement Exercise 6.3 using NumPy arrays

```
import numpy as np
import random
import sys

def dice2(N):
    '''Non-vectorized, draw numbers in a loop and count successes'''
    s = 0
    for experiment in range(N):
        results = [random.randint(1,6) for i in range(2)]
        if 6 in results:
            s += 1
    return s/float(N)

def dice2_vec(N):
    '''Vecotrized, draw all numbers at once and extract successes'''
    # Draw random integers
    r = np.random.randint(1, 7, (N, 2))
    # Create a boolean array
    s = (r[:,0] == 6) | (r[:,1] == 6)
    # Sum all successes
    c = np.sum(s)
    # Calculate and return probability
    return c/float(N)


if __name__ == '__main__':
    # Time the functions
    import timeit

    # The setup is used by timeit, but does not contribute to the final time
    setup = "from __main__ import N, dice2, dice2_vec"
    print "Run-time for functions\n%5s%10s%12s" % ('N', 'dice2', 'dice2_vec')
    for N in [10**3, 10**4, 10**5, 10**6, 10**7]:
        t1 = timeit.timeit('dice2(N)', setup=setup, number=1)
        t2 = timeit.timeit('dice2_vec(N)', setup=setup, number=1)
        print "%5.0e%10.2e%12.2e" % (N, t1, t2)

'''
user$ python dice2_NumPy.py
Run-time for functions
    N     dice2   dice2_vec
1e+03  2.98e-03    1.66e-04
1e+04  2.85e-02    5.75e-04
1e+05  2.84e-01    5.18e-03
1e+06  2.90e+00    6.17e-02
1e+07  2.89e+01    6.03e-01
'''
```

7.6—Implement Exercise 6.4 using NumPy arrays

```
import numpy as np
import random
import sys

def dice4(n):
    '''Non-vectorized, draw numbers in a loop and count successes'''
    money = 0
    for _ in range(n):
        money -= 1
        die = [random.randint(1,6) for _ in range(4)]
        if sum(die) < 9:
            money += 10
    return money/float(n)

def dice4_vec(n):
    '''Vecotrized, draw all numbers at once and extract successes'''
    # Draw random integers and reshape
    r = np.random.randint(1,7, 4*n).reshape((n,4))
    # Could also have simply done
    #r = np.random.randint(1,7,(n,4))

    # Sum the 4 integers for every game
    s = np.sum(r, axis=0)
    # Find the winning results
    wins = s[s<9]
    # Calculate total winnings
    money = 10*len(wins) - n
    return money/float(n)

if __name__ == '__main__':
    # Time the functions
    import timeit

    # The setup is used by timeit, but does not contribute to the final time
    setup = "from __main__ import n, dice4, dice4_vec"
    print "Run-time for functions\n%5s%10s%12s" % ('n', 'dice4', 'dice4_vec')
    for n in [10**3, 10**4, 10**5, 10**6, 10**7]:
        t1 = timeit.timeit('dice4(n)', setup=setup, number=1)
        t2 = timeit.timeit('dice4_vec(n)', setup=setup, number=1)
        print "%5.0e%10.2e%12.2e" % (n, t1, t2)

'''
user$ python dice4_NumPy.py
Run-time for functions
    n     dice4   dice4_vec
1e+03  5.70e-03    1.83e-04
1e+04  5.73e-02    9.87e-04
1e+05  5.73e-01    9.64e-03
1e+06  5.76e+00    1.29e-01
1e+07  5.78e+01    1.29e+00
'''
```