Object-oriented programming - Day 2

March 23, 2015

1 Simple class - String

1. Write a class `String` which consists of only one method, the constructor. Apart from the obligatory `self`, the constructor shall take one additional argument `s`. In the constructor initialize a member variable of the class, let's call it `mystring`, with the value of `s`.
2. Add an additional method to the class. This method shall also take one argument called `other` (+ `self`). In the function body check if `other` is contained in `mystring`. If so return `True`, otherwise return `False`. *Hint*: Try the `in` keyword.
3. Add a class (`static`) variable called `counter` to the class and initialize it with 0.
4. Change the initializer such that it increments `counter` by one.
5. Create several instances of this `String` class and check afterwards how many you have.
6. Add a 'destructor' function `__del__` which decreases `counter` by one. Test it, i. e. inspect the value of counter.

In [2]:
```
class String:
    """Checks if a given string is contained in the member string"""
    counter = 0
    def __init__(self, mystring):
        self.mystring=mystring
        String.counter += 1

    def __del__(self):
        String.counter -= 1

    def check(self, other):
        if other in self.mystring:
            return True
        else:
            return False

def test_string():
    s1=String("Hello")
    s2=String("World")

    print(s1.check("el"), s2.check("el"))

    print("There are %u String instances" % String.counter)

    del s2
    print("Now: %u String instances" % String.counter)

    s1=1
    print("And now: %u" % String.counter)
```

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

```
True False
There are 2 String instances
Now: 1 String instances
And now: 0
```

2 Inheritance - Animal

1. Write an `Animal` base class. Each animal shall have an `age` and a `weight`. Once set in the constructor these variables should not be changeable from the outside (you cannot just change the age of an animal as you like, right?). But provide functions that allow to read the values of these variables. *Note:* In Python "private" attributes are not really private. But it is the best protection of member attributes against the outside that we can get — so we take it.

2. Implement a `speak` and a `move` method which print an error message (an abstract animal can neither speak nor move).

3. Write a `Cat` and a `Fish` class which inherit from `Animal`. Override the `speak` and `move` methods such that they print out an appropriate message when called (something like "Meow" for the cat. . .)

4. Test your implementation in the python interpreter by creating instances of `Cat` and `Fish`.

5. Try to make sense of what's happening here:

   ```
   > c = Cat(2, 3.7)
   > c.speak( )
   Meow.
   > Animal.speak(c)
   I am an abstract animal and cannot speak.
   > num = 3.7
   > Animal.speak(num)
   [some peculiar Python−error message]
   ```

6. Add an `eat` method which takes as an argument the object to eat; for exam- ple: `mycat.eat(myfish)`. Obviously the fish should be dead afterwards, so the method's purpose is to "kill" the fish, i.e. the fish should not be accessible any more (the `myfish` instance should be deleted). How could you accomplish this?

In [3]:
```
class Animal:
    """An abstract animal"""

    def __init__(self, weight, age):
        # store your own copy of weight and age such that we can
        # refer to them later
        self.__weight=weight
        self.__age = age

        # look at move() function
        self._weightLossPerDistance = 1.0

    def age(self):
        """ This function allows to access self.__age from outside """
```

```
        return self.__age

    def weight(self):
        """ This function allows to access self.__weight from outside """
        return self.__weight

    def eat(self, amount):
        """ When the animal eats, it inceases the weight by amount. """
        self.__weight += amount

    def move(self, distance):
        """ Motion needs energy, therefore this function decreases the
        weight according to the constant self.__weightLossPerDistance.
        If self.__weight falls below 0 an error message is printed.
        """
        self.__weight -= self._weightLossPerDistance*distance
        if self.__weight<=0.0:
            print ("I'm starving...")

    def speak(self):
        print ("I am an abstract animal which can't speak")

class Fish(Animal):
    """This is a fish"""

    def __init__(self, weight, age):
        # Call the constructor of the parent class
        Animal.__init__(self, weight, age)

    def move(self, distance):
        # Use the behaviour from the parent class, and, in addition,
        # print a fish-specific message
        Animal.move(self, distance)
        print ("I'm swimming through the aquarium")

    def speak(self):
        print ("blub")

class Cat(Animal):
    """I'm a cat"""

    def __init__(self, weight, age):
        Animal.__init__(self, weight, age)
        self._weightLossPerDistance = 3.0

    def move(self, distance):
        Animal.move(self, distance)
        print ("I'm chasing mice")

    def speak(self):
        print ("Miau")

    def eat(self, fish, animalDi):
        """
```

```
            This function checks if the argument 'fish' really is of type Fish
            and, if so, eats it. This accounts to deleting it from the dictionary
            'animalDi'. Note that it is (as far as I know) not possible to
            globally erase the fish instance from memory. So deleting it from
            a "reservoir of animals" is the closest thing to killing it.
            """
            if isinstance(fish,Fish):
                delvar=None
                for a in animalDi:
                    if id(fish)==id(animalDi[a]):
                        delvar=a
                        break
                if delvar!=None:
                    del animalDi[a]
                    print ("Hmmm, that was delicious. Rest in peace fish!")
                else:
                    print ("Couldn't see the fish to eat")
            else:
                print ("I'm a cat, I don't eat that crap")

    def test_animal():
        a = Animal(27.23,1)
        f = Fish(3.141,2)
        c = Cat(2.718,3)

        pets = {
            'abstract': a,
            'neo': f,
            'snowball': c
            }

        for p in pets:
            pets[p].speak()
            pets[p].move(2)

        c.eat(a,pets)
        c.eat(f,pets)
        try:
            pets['neo'].speak()
        except(KeyError):
            print ("Sorry, the fish is gone")

        # but accessing f directly still works:
        f.speak()

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

```
blub
I'm swimming through the aquarium
Miau
I'm starving...
I'm chasing mice
I am an abstract animal which can't speak
```

```
I'm a cat, I don't eat that crap
Hmmm, that was delicious. Rest in peace fish!
Sorry, the fish is gone
blub
```

3 An integrate-and-fire neuron

A more complex example: using what we've learned so far

The equation determining the membrane potential of a *leaky integrate and fire neuron* is given by

$$c_M \dot{V} = -g_L (V - V_L) + i_{ext}$$

where c_M is the membrane potential, g_L the leak conductance, V_L the corresponding reversal potential and i_{ext} an external current that drives the neuron. In addition, whenever V becomes larger than a threshold value V_{th} a spike is elicited and V is reset to a value V_r.

1. Write a class which is initialized with the necessary parameters (like $c_M, \ldots,$ don't forget an initial value for V). Except for the external current it should not be possible to change the neuron parameters after instanciation. Cf. the hint in exercise 2.1.

2. Using the Euler method
   $$\dot{V} \approx \frac{V(t+h) - V(t)}{h}$$
   we can derive the following update rule:
   $$\begin{aligned} V(t+h) &= V(t) + h\dot{V}(t) && (1) \\ &= V(t) + \frac{1}{c_m}\left(-g_L (V - V_L) + i_{ext}\right) && (2) \end{aligned}$$
   Implement a method which updates V according to this rule until a time T has passed. In each step append the newly calculated V to a list `trace`.

3. Now consider that V is reset everytime it crosses the threshold.

4. Add a method that uses *matplotlib* to plot the voltage trace. As *matplotlib* will be treated later, here's a spoiler: import matplotlib.pyplot as plt plt.plot(xvals,yvals)

5. Test your neuron for different initial values for V and different external currents i_{ext}.

6. *If you are quick:* Simulate a network of neurons. Each neuron is connected to some others (e.g. with a predefined probability). The equation to simulate is now
   $$c_M \dot{V}_j = -g_L (V_j - V_L) + i_{ext} + i_{j,syn}$$
   where
   $$i_{j,syn} = \sum_k \sum_m w_{jk} K(t - t_{jk}^{(m)})$$
   and $K(t) = \delta_{t,0}$. $w_{j,m}$ are weight factors specifying the strength of the connection between neuron m and j and $t_{jk}^{(m)}$ is the time when neuron j receives its m-th spike from neuron k. In other words: whenever neuron j receives a spike from neuron k its voltage changes by an amount w_{jk}.

In [26]:
```
from numpy import *
import matplotlib.pyplot as plt

class IFNeuron:
    """Integrate-and-fire neuron"""

    def __init__(self, cm=1, gL=0.01 , EL=-55, Vthreshold=0, Vreset=-70, V0=-60,
    stepSize = 0.1):
        # Make local copies of all parameters
        # For simplicity, they are all public variables
        # In a more 'restrictive' version, I would make them private
        # (i.e. replace self.cm by self.__cm and likewise for the others)
        # and provide functions to access and, maybe, change them
        self.cm = cm
        self.gL = gL
        self.EL = EL
        self.Vthreshold = Vthreshold
        self.Vreset = Vreset
        self.Vtrace = [V0]
        self.stepSize = stepSize

        self.Iext = 0.0

        self.__calcODEConsts()


    def __calcODEConsts(self):
        # These constants are needed in the __step function
        self.tau = self.cm/self.gL
        self.Vinf = self.EL + float(self.Iext)/self.gL

    def setIext(self, Iext):
        """ Set external current received by neuron """
        self.Iext = Iext
        # as self.tau and self.Vinf depend on Iext, they have to be
        # recalculated
        self.__calcODEConsts()

    def __step(self):
        # Evolves membrane potential of this neuron by one timestep
        Vdot = float(-self.Vtrace[-1]+self.Vinf)/self.tau
        V = self.Vtrace[-1] + self.stepSize * Vdot #explicit Euler method
        self.Vtrace.append(V)

        # If membrane potential crosses threshold: fire!
        if V>=self.Vthreshold:
            self.__fire()

    def __fire(self):
        # For an integrate-and-fire neuron, firing only means to reset
        # the voltage
        self.Vtrace[-1] = self.Vreset
        # One could, in principal, also record the time of the spike
```

```python
    def run(self, T):
        """ Evolve the neuron for a time T """
        t=0
        while t<T:
            self.__step()
            t += self.stepSize

    def plot(self):
        """ Plot voltage trace as far as it has been simulated already """
        plt.plot(self.Vtrace)
        plt.xlabel('Time (ms)')
        plt.ylabel('Voltage (mV)')
        plt.show()

def test_ifneuron():
    n = IFNeuron()
    n.setIext(1)
    n.run(1000)
    n.plot()

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

In [29]:
```python
from numpy import *
import matplotlib.pyplot as plt

class IFNeuron:
    """Integrate-and-fire neuron
       =========================
       This version of IFNeuron is, essentially, idential to the one
       in file 'ifneuron.py'. Only some details changed that were necessary
       or convenient if the neuron is supposed to be part of a network of
       neurons.
    """

    def __init__(self,
                 neuronID,
                 network,
                 cm=1,
                 gL=0.01 ,
                 EL=-55,
                 Vthreshold=0,
                 Vreset=-70,
                 V0=-60,
                 stepSize = 0.1,
                 ):

        # These are new in the network version
        self.__neuronID = neuronID
        self.__network = network

        self.cm = cm
        self.gL = gL
        self.EL = EL
```

```python
        self.Vthreshold = Vthreshold
        self.Vreset = Vreset
        self.Vtrace = [V0]
        self.stepSize = stepSize

        self.Iext = 0.0

        self.__calcODEConsts()

    def __calcODEConsts(self):
        # For convenience, store these combinations of the variables
        # because it's exactly these combinations that occur in the
        # step(self) function
        self.tau = float(self.cm)/self.gL
        self.Vinf = self.EL + float(self.Iext)/self.gL

    def setIext(self, Iext):
        """ Set external current impinging on this neuron. """
        self.Iext = Iext
        self.__calcODEConsts()

    def step(self):
        """ Evolve voltage of this neuron by one time step.
        Compared to the non-network version, it is now a public function,
        because it is supposed to be called by the network, that is: from
        outside the neuron.
        """
        Vdot = float(-self.Vtrace[-1]+self.Vinf)/self.tau
        V = self.Vtrace[-1] + self.stepSize * Vdot #explicit Euler method
        self.Vtrace.append(V)

        # The part checking for a spike is now a separate function

    def checkForSpike(self):
        """ If voltage crosses threshold, then fire """
        if self.Vtrace[-1]>=self.Vthreshold:
            self.__fire()

    def __fire(self):
        """ If this neuron fires, send a spike to all connected neurons,
            that is, send a spike causing a psp to all neurons, only if
            psp != 0 this has an effect """
        self.Vtrace.append(self.Vreset)
        for i in range(self.__network.nNeurons()):
            psp = self.__network.psp(i,self.__neuronID)
            self.__network.neuron(i).receiveSpike(psp)

    def receiveSpike(self,psp):
        """ The effect of receiving a spike is to change the voltage of
            this neuron by an amount psp
        """
        self.Vtrace[-1] += psp

    def plot(self):
```

```
        """ Plot voltage trace """
        plt.plot(self.Vtrace)
        plt.show()

import numpy as np
from random import random

class NeuralNetwork:
    """Neural network class
       ====================
    """

    def __init__(self, nNeurons):

        self.__nNeurons = nNeurons

        self.__neurons = []
        for i in range(nNeurons):
            n = IFNeuron(i,self)
            n.setIext(1)
            self.__neurons.append(n)

        # connectionMatrix is a matrix of size nNeurons x nNeurons
        # Its entries c_{ij} give the post-synaptic-potential neuron i
        # receives if neuron j spikes.
        # I.e. neuron j has a synapse on neuron i if and only if
        # c_{ij} != 0
        # Technical note: I use a numpy array to store the matrix,
        # but in principle one could also use a list of lists
        self.__connectionMatrix = np.zeros( (nNeurons,nNeurons) )

        # interconnect neurons with probability 0.25
        self.__connectNeurons(0.25)

    def __connectNeurons(self,p):
        """ Connect all neurons in network to all others with probability p
            For simplicity assume that all PSPs are the same
        """

        psp = 1

        for i in range(self.__nNeurons):
            for j in range(self.__nNeurons):
                if random()<p:
                    self.__connectionMatrix[i,j] = psp

    def nNeurons(self):
        """ Returns the number of neurons in the network """
        return self.__nNeurons

    def psp(self,i,j):
        """ Returns the postsynaptic-potential in neuron i when receiving
            a spike from neuron j
        """
```

```
            return self.__connectionMatrix[i,j]

        def neuron(self,i):
            """ Return neuron number i in the network """
            return self.__neurons[i]

        def simulate(self,nTimesteps):
            """ Run the simulation for nTimesteps steps """

            for i in range(nTimesteps):
                # first, evolve voltage
                for n in self.__neurons:
                    n.step()

                # then, check which neuron spiked and, if so, transmit spikes
                for n in self.__neurons:
                    n.checkForSpike()

    def test_network():
        nn = NeuralNetwork(100)
        nn.simulate(10000)
        nn.neuron(0).plot()
        nn.neuron(1).plot()

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

4 The Zoo

Dictionaries, special methods, random numbers, functions as arguments to functions

We want to collect some animals (the ones from the `Animal` exercise above) in a `Zoo` object. It will consist of a number of animals and each of them should have a name. 1. Provide a way to add new animals to the `Zoo`. 2. *Special methods.* Implement some special methods like - the length-operator `len(self)`, to see how many inhabitants the zoo has. Test with `len(myzoo)` (where `myzoo` is an instance of `Zoo`) - the "less than" operator `lt(self,rhs)` which should return `True` if `self<rhs` and `False` if `self>=rhs`. Test with `myzoo1` < `myzoo2` ... - the subscripting operator `getitem(self, name)` to access an animal from the zoo by name (e.g. `z["blub"].speak()`), - ... 3. The zoo welcomes 1000 new animals. As the administration can't come up with so many names at the same time, they shall be named, for the moment, by numbers. Write a method which adds a number `n` of new animals of different species, age and weight. You can use the functions `random, randint` and `randrange` from module `random` to create random numbers. 4. Make it possible to rename animals. 5. For easy book-keeping add a `select` method which takes one argument `fltr`. `fltr` is itself a function accepting an `Animal` instance as argument and returning a boolean. `select` returns a list of all animals for which the `fltr` function returns `True`. Test this function: - select all animals which are younger than 2 - select all animals for which `age+weight` $\leq \pi$ - select all `Cats` 6. Let the zoo visitor specify a simple(!!!) criterium for animal selection. I. e. create a method `visitorSelect` with a string input argument which evaluates it and then calls `select`.

```
In [16]: class Zoo:
             """This zoo collects animals which have a name.
             Internally they are stored in a dictionary.
             """
             def __init__(self):
                 # Initialize with 0 animals
                 self.inhabitants = {}
```

```
def __len__(self):
    return len( self.inhabitants )

def __lt__(self, rhs):
    if not isinstance(rhs, Zoo):
        print ("I can only compare Zoos.")
        return

    mylen = len(self)
    rhslen = len(rhs)
    if mylen<rhslen : return 1
    else: return 0

def __gt__(self, rhs):
    if not isinstance(rhs, Zoo):
        print ("I can only compare Zoos.")
        return

    mylen = len(self)
    rhslen = len(rhs)
    if mylen>rhslen : return 1
    else: return 0

def add(self,rhs):
    #check if rhs has correct format
    if not isinstance(rhs, tuple):
        print ("Argument is no tuple.")
        return
    if not isinstance(rhs[0], str):
        print ("First part of tuple must be a string.")
        return
    if not isinstance(rhs[1], Animal):
        print ("Second part of tuple must be an animal.")
        return

    self.inhabitants[ rhs[0] ] = rhs[1]

def addMany(self, Species, nAnimals):
    from random import random,randint
    for i in range( len(self.inhabitants),len(self.inhabitants)+nAnimals ):
        self.inhabitants[str(i)] = Species(10*random() # weight
                                           , randint(0,50) # age
                                           )
def rename(self, oldName, newName):
    self.inhabitants[newName] = self.inhabitants[oldName]
    del self.inhabitants[oldName]

def select(self, fltr):
    """
    Examples for fltr:
    - lambda x: x.age()<2
    - lambda x: x.age()+x.weight()<3.142
    - lambda x: isinstance(x, Cat)
```

```
        """

        inh = self.inhabitants # abbreviation
        return [ inh[x] for x in inh if fltr(inh[x]) ]

    def visitorSelect(self, inp):
        return self.select( lambda x: eval(inp) )

    def __getitem__(self,name):
        return self.inhabitants[ name ]

def test_zoo():
    z1=Zoo()
    z1.add( ("snowball", Cat(3.14,3)) )

    z2=Zoo()
    z2.add( ("salmon", Fish(1.2,1)) )
    z2.add( ("jellyfish", Fish(2.7,2)) )

    print ("Second zoo has %u inhabitants." % len(z2))
    print ("Is first zoo smaller than second zoo?", z1<z2)

    z1["snowball"].speak()

    z1.addMany(Cat, 100)
    z1.addMany(Fish,100)
    z1.rename("1","icecube")

    youngAnimals = z1.select(lambda x: x.age()<2)
    weirdAnimals = z1.select(lambda x: x.age()+x.weight()<=3.14)
    allCats = z1.select(lambda x: isinstance(x, Cat) )
    print (len(youngAnimals), len(weirdAnimals), len(allCats))

    visitorsAnimals = z1.visitorSelect('x.age()<2')
    print ("You visit %u animals." % len(visitorsAnimals))

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

```
Second zoo has 2 inhabitants.
Is first zoo smaller than second zoo? 1
Miau
7 3 101
You visit 7 animals.
```

5 Temperature

Write a class that stores a temperature in one unit and allows accessing it in several other ones (cf. http://en.wikipedia.org/wiki/Conversion_of_units_of_temperature for conversion formulas).

Hint: Have a look at `__getattr__` , `__setattr__` to access and set temperatures.

In [20]:
```
## Source: {{{ http://code.activestate.com/recipes/286226/ (r2)

class Temperature(object):
```

```
    equations = {'c': (1.0, 0.0, -273.15), 'f': (1.8, -273.15, 32.0),
                 'r': (1.8, 0.0, 0.0)}

    def __init__(self, k=0.0, **kwargs):
        """ Allow the temperature instance to be created as:
        temp = Temperature() # to initialize with 0 Kelvin
        temp = Temperature(k=x) # to initialize with x Kelvin
        temp = Temperature(c=x) # to initialize with x Celsius
        temp = Temperature(f=x) # to initialize with x Fahrenheit
        """
        self.k = k
        for k in kwargs:
            if k in ('c', 'f', 'r'):
                setattr(self, k, kwargs[k])
                break

    def __getattr__(self, name):
        """ Let temp be an instance of this Temperature class.
        This function allows to access the temperature values in all units
        (K,C,F) using dot-notation. I.e. temp.k returns temperature in Kelvin,
        temp.c in Celsius and temp.f in Fahrenheit
        """
        if name in self.equations:
            eq = self.equations[name]
            return (self.k + eq[1]) * eq[0] + eq[2]
        else:
            return object.__getattribute__(self, name)

    def __setattr__(self, name, value):
        """ Similar to __getattr__, but this function allows to set
        temperatures using dot-notation, for example:
        temp.k=10 to set temperature to 10K.
        """
        if name in self.equations:
            eq = self.equations[name]
            self.k = (value - eq[2]) / eq[0] - eq[1]
        else:
            object.__setattr__(self, name, value)

    def __str__(self):
        return "%g K" % self.k

    def __repr__(self):
        return "Temperature(%g)" % self.k
## end of http://code.activestate.com/recipes/286226/ }}}

def test_tempconv():
    print ("Setting temperature to 10K")
    temp = Temperature(k=10)
    print (temp.k, 'K =', temp.c, 'C =', temp.f, 'F')

    print ("Setting temperature to 5C")
    temp.c = 5
    print (temp.k, 'K =', temp.c, 'C =', temp.f, 'F')
```

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

```
Setting temperature to 10K
10 K = -263.15 C = -441.66999999999996 F
Setting temperature to 5C
278.15 K = 5.0 C = 41.0 F
```

6 Vector

A `list` or `tuple` can be used to store floating point numbers. They are, however, not suitable as vector classes. Namely, apart from performance issues, it would be desirable if one could do basic calculations, such as addition by writing `z = x + y`, where `x, y, z` are instances of such a vector class. (Note that a `list` has an operator + but it does something else!). Therefore write a `Vector` class which allows basic vector space operations (e. g.: addition, subtraction, scalar multiplication and division, dot product, . . .).

```
In [52]: from math import sqrt, acos, degrees, radians, cos, sin, fsum, hypot, atan2


        class Vector(tuple):

            '''"Class to calculate the usual operations with vectors in bi and
            tridimensional coordinates. Too with n-dimmensinal.'''
            # __slots__=('V') #It's not possible because V is a variable list of param.
            def __new__(cls, *V):
                '''The new method, we initialize the coordinates of a vector.
                You can initialize a vector for example: V = Vector() or
                V = Vector(a,b) or V = Vector(v1, v2, ..., vn)'''
                if not V:
                    V = (0, 0)
                elif len(V) == 1:
                    raise ValueError('A vector must have at least 2 coordinates.')
                return tuple.__new__(cls, V)

            def __add__(self, V):
                '''The operator sum overloaded. You can add vectors writing V + W,
                where V and W are two vectors.'''
                if len(self) != len(V):
                    raise IndexError('Vectors must have same dimensions.')
                else:
                    added = tuple(a + b for a, b in zip(self, V))
                    return Vector(*added)

            def __sub__(self, V):
                '''The operator subtraction overloaded. You can subtract vectors writing
                V - W, where V and W are two vectors.'''
                if len(self) != len(V):
                    raise IndexError('Vectors must have same dimensions.')
                else:
                    subtracted = tuple(a - b for a, b in zip(self, V))
                    return Vector(*subtracted)

            def __mul__(self, V):
```

```
        '''The operator mult overloaded. You can multipy 2 vectors coordinate
         by coordinate.'''
        if type(V) == type(self):
            if len(self) != len(V):
                raise IndexError('Vectors must have same dimensions')
            else:
                multiplied = tuple(a * b for a, b in zip(self, V))
        elif isinstance(V, type(1)) or isinstance(V, type(1.0)):
            multiplied = tuple(a * V for a in self)
        return Vector(*multiplied)

    def __truediv__(self, V):
        if type(V) == type(self):
            if len(self) != len(V):
                raise IndexError('Vectors must have same dimensions.')
            if 0 in V:
                raise ZeroDivisionError('Division by 0.')
            divided = tuple(a / b for a, b in zip(self, V))
        elif isinstance(V, int) or isinstance(V, float):
            divided = tuple(a / V for a in self)
        return Vector(*divided)

    def __pow__(self, n):
        '''The operator pow overloaded. You can powering vectors writing
         V ** n, where V is a vector, and n is the power. If V = (a, b), then
         V ** n calculates (a ** n, b ** n)'''
        powered = tuple(a ** n for a in self)
        return Vector(*powered)

    def __neg__(self):
        '''The operator negative overloaded. If V is a vector, you can
        calculate -V, the vector V changed its sign in its coordinates.'''
        opposite = tuple(-1 * a for a in self)
        return Vector(*opposite)

    def norm(self):
        '''Returns the norm (length, magnitude) of the vector'''
        return sqrt(fsum(comp ** 2 for comp in self))

    def normalize(self):
        '''Returns a normalized unit vector'''
        norm = self.norm()
        normed = tuple(comp / norm for comp in self)
        return Vector(*normed)

    def inner(self, V):
        ''' Returns the dot product (inner product or scalar product) of self
        and V vector
        '''
        return fsum(a * b for a, b in zip(self, V))

def test_vector():
    x = Vector(1, 2, 3, 4)
    y = Vector(1, 1, 1, 1)
```

```
        z = Vector(0, 0, 0, 0)
        print("x = ", x)
        print("y = ", y)
        print("z = ", z)
        print ("x - y = ", (x-y))
        print ("y/|y| = ", (y.normalize()))
        #print ("x / z = ", (x/z))

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

```
x =  (1, 2, 3, 4)
y =  (1, 1, 1, 1)
z =  (0, 0, 0, 0)
x - y =  (0, 1, 2, 3)
y/|y| =  (0.5, 0.5, 0.5, 0.5)
```

In []: