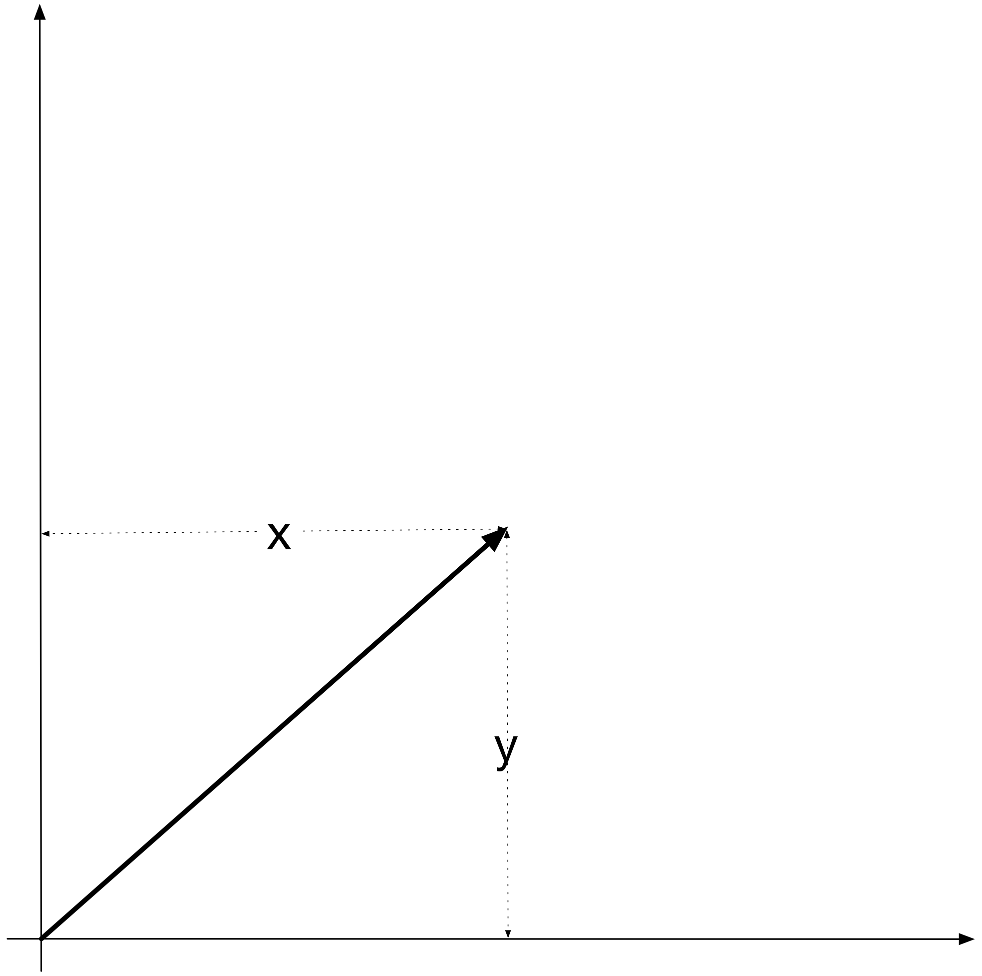
Repetition
Classes

TwoDVector Class

Thomas Schwarz, SJ

TwoDVector

- TwoDVector:
    - encapsulates two components



TwoDVector

- Define all the components of a class in the constructor (__init__ dunder)

```
class TwoDVector:
    def __init__(self, a, b):
        self.x = a
        self.y = b
```

TwoDVector

- The str and repr dunders have only self as argument and return a string
- Usually, we use format

```
def __str__(self):
        return '({},{})'.format(self.x, self.y)
    def __repr__(self):
        return '<Vector self.x = {}, self.y = {}
>'.format(self.x, self.y)
```

TwoDVector

- The str dunder is called when we communicate with the end user, for example in a print statement

```
>>> a = TwoDVector(4,5)
>>> print(a)
(4,5)
```

TwoDVector

- The repr dunder is used by the programmer
    - Usually contains more information
    - Will be returned by the shell

```
>>> TwoDVector(2,3)
<Vector self.x = 2, self.y = 3>
>>>
```

Overwriting functions

- Python has several standard functions that can be overwritten
  - __len__( )   for len( )
  - __abs__( ) for abs( )
- The absolute value of a vector is the square-root of the sum of squares of its coordinates

```
def __abs__(self):
        return math.sqrt(self.x**2+self.y**2)
```

Overwriting functions

- We call the absolute-value function using the abs function on an object of the class

```
>>> myvector = TwoDVector(-3,2)
>>> abs(myvector)
3.605551275463989
>>>
```

Overwriting Operators

- Python numbers can be compared with <, >, <=, >=, ==, and !=
- We can implement these Boolean operators using dunder functions with magic names.
- In the context of a vector, comparisons other than for equality and inequality make no sense

Overwriting Operators

- It is traditional to use self and other as the names for the operands

```
def __eq__(self, other):
     return self.x==other.x and self.y==other.y
def __ne__(self, other):
     return self.x!=other.x or self.y!=other.y
```

Overwriting Operators

- Python allows us to define our own versions of arithmetic operators
- We can say

```
>>> b = TwoDVector(2,1)
>>> a = TwoDVector(4,5)
>>> a+b
<Vector self.x = 6, self.y = 6>
```

Overwriting Operators

- To create an addition, we use the __add__ dunder
- We need to return the result as a new object
  - In our case, creating a new Vector object

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

Overwriting Operators

- To use the += operator, we overwrite __iadd__
  - This one modifies self, but also returns self

```
def __iadd__(self, other):
        self.x += other.x
        self.y += other.y
        return self
```

Overwriting Operators

```
>>> b = TwoDVector(2,1)
>>> a = TwoDVector(4,5)
>>> a+b
<Vector self.x = 6, self.y = 6>
>>> a += b
>>> print(a)
(6,6)
>>>
```

Overwriting Operators

- Selftest:
    - Implement and try out subtraction

Overwriting Operators

- Vectors do not have a traditional multiplication, but they have the "dot" product
  - $(a, b) \cdot (c, d) = ac + bd$
- The result of the dot multiplication is a scalar, not another vector
- But this is no problem, we just return a scalar for __mul__

Overwriting Operators

```
def __mul__(self, other):
    return self.x*other.x+self.y*other.y
```

- Usage is no different then for normal operations

```
>>> a = TwoDVector(4,5)
>>> b = TwoDVector(2,1)
>>> a*b
13
```

Overwriting Operators

- When Python encounters an expression a+b
  - It first checks whether there is an __add__ dunder for a
  - Then it checks whether there is an __radd__ dunder for b
  - It is not necessary that a or b are objects of the same class

Overwriting Operators

- Vectors have a scalar multiplication
  - $x \cdot (a, b) = (xa, xb)$
- Notice that we traditionally write the scalar to the left and the vector to the right
- So we can use __rmul__ to implement scalar multiplication
  - We still need to return a new Vector

```
def __rmul__(self, nr):
     return TwoDVector(self.x*nr, self.y*nr)
```

Overwriting Operators

- If we try `a *= 3` for a vector a, the __mul__ operation gets invoked, effectively calculating `a = a*3`
  - This results in an error, since an integer does not have x and y fields
- However, we can implement __imul__ and then it will work

```
def __imul__(self, other):
     self.x *= other
     self.y *= other
     return self
```

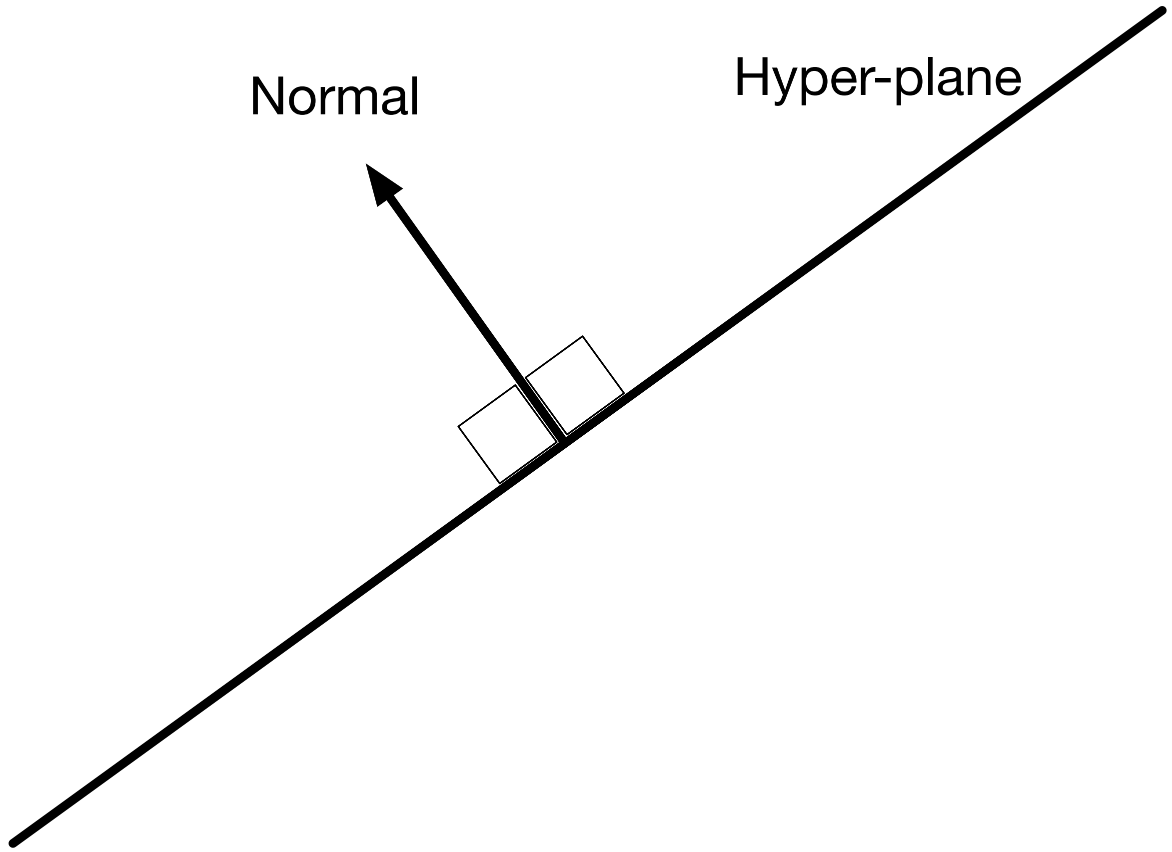
Implementing Functions

- Two-dimensional vectors have a rich set of functions
- We can rotate a vector by an angle θ using the formula:
- $$\begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix}$$

```
def rotate_by(self, theta):
   return TwoDVector(self.x*math.cos(theta)-self.y*math.sin(theta),
                     self.x*math.sin(theta)+self.y*math.cos(theta))
```

Implementing Functions

- A reflection on a hyperplane is defined in terms of a normal, a vector standing orthogonal on the plane



Implementing Functions

- The formula for the reflection uses the dot-product
- a is the normal of the hyperplane
- $$v \longrightarrow v - 2\frac{v \cdot a}{a \cdot a}a$$

Implementing Functions

```
def reflect_by(self, other):
    '''Reflection of self on hyperplane orthogonal to other'''
    return self - 2*(self*other)/(other*other)*other
```