Python

and

Protocols

Univ.AI

Polymorphism

```python
from math import pi

class Circle(Shape):

    def __init__(self, r):
        super().__init__("circle")
        self.r = r

    def area(self):
        return pi*self.r*self.r

class Square(Shape):

    def __init__(self, side):
        super().__init__("square")
        self.side = side

    def area(self):
        return self.side*self.side
```

```python
class Shape: # Shape(object)

    def __init__(self, name):
        self.name = name

    def area(self):
        raise NotImplementedError

c = Circle(1)
s = Square(2)
shapes = [c, s]
total_area = 0
for shp in shapes:
    total_area += shp.area()
total_area # 7.141592653589793
```

You define a *protocol* as to how classes will behave. Here, all shapes *must* have an `area` method with no arguments. Then you can use this protocol without regard for the underlying shape type, as in calculating the total area.

Vectors in Computer Science

We will call this a 2D vector, or a 2D **column** vector. The numbers might represent anything, for example, the properties of a house: 1 (in units of 1000) square feet and 2 bedrooms.

```
v = Vector([1, 2])

len(v) # returns 2
v[0] # returns 1
v[1] # returns 2
```

For us, thus, a vector is an ordered list of numbers. We'll use this to define a `Vector` class which has a length and *components*:

```
class Vector:

    def __init__(self, lst):
        self.storage = lst.copy()

    def __len__(self):
        return len(self.storage)

    def __getitem__(self, i):
        return self.storage[i]
```

Python: protocols on steroids

```python
class Vector:

    def __init__(self, lst):
        self.storage = lst.copy()

    def __len__(self):
        return len(self.storage)

    def __getitem__(self, i):
        return self.storage[i]

v = Vector([3, 4, 5, 6, 7])
# Length of Vector instance!!
len(v) # 5
# The Vector instance supports indexing!!
v[1] # 4
```

The class `Vector` adheres to something called the *sequence protocol*. It does not need to inherit from a list, instead using a list as its storage. But it **must** implement `__len__` (called dunder len) and `__getitem__` (dunder getitem). It implements these by *delegating* to the corresponding functions in the underlying list storage. By implementing `__len__`, `Vector` can respond to the built in `len` function, and by implementing `__getitem__`, it can be indexed. *Python is built on this notion of responding to protocols by implementing dunder functions!*

Dunder Methods

- we saw that methods like `__len__` are what a python type or class needs to implement to respond to built in functions like `len`
- these are implemented for us for the built-in types such as lists, tuples, dictionaries, etc, but we can also implement out own as we did in `Vector`.
- this implementing of *dunder methods* constitutes the Python Data Model

```
mylist = [1,2,3]
len(mylist) # 3
mylist[1] # 2

mytuple = (1,2,3)
len(mytuple) # 3
mytuple[1] # 2

mydict = {1:1, 2:4, 3:9}
len(mydict) # 3
for k in mydict:
    print(k) # print key
# 1, 2, e : BUT dicts may not give keys in
# the order in which they were defined
mydict[2] # 4, this is actually a
# key lookup, not an indexing
```

Dunder methods do a lot!

Before:

```
v1 = Vector([4, 2])
v1 # <__main__.Vector at 0x112966710>
```

To implement nice printing,

```
class Vector:

    def __init__(self, lst):
        self.storage = lst

    def __len__(self):
        return len(self.storage)

    def __getitem__(self, i):
        return self.storage[i]

    def __repr__(self):
        return f"Vector({self.storage})"

v1 = Vector([3, 4, 5, 6])
v1 # Vector([3, 4, 5, 6])
```