Lecture 6 Class and Modules

A possibly overlooked point: Modules and Class in Python share many similaries at the basic level. They both define some names (attributes) and functions (methods) for the convenience of users -- and the codes to call them are also similar. Of course, Class also serves as the blue prints to generate instances, and supports more advanced functions such as Inheritance.

Class and Instance

Simple Example of Vector

Let's first define the simplest class in Python

In [ ]:
```
class VectorV0:
    '''The simplest class in python'''  # this is the document string

    pass
```

and create two instances `v1` and `v2`

In [ ]:
```
v1 = VectorV0()  # note the parentheses here
v2 = VectorV0()
```

Now `v1` and `v2` are the objects in Python

In [ ]:
```
type(v1)
```

In [ ]:
```
dir(v1)
```

We can manually assign the attributes to instance `v1` and `v2`

In [ ]:
```
v1.x = 1.0
v1.y = 2.0
v2.x = 2.0
v2.y = 3.0
```

In [ ]:
```
dir(v1)
```

We don't want to create the instance or define the coordinates seperately. Can we do these in one step, when initializing the instance?

In [ ]:
```
class VectorV1:
    '''define the vector'''  # this is the document string
    dim = 2   # this is the attribute
    def __init__(self, x=0.0, y=0.0):  # any method in Class requires the first parameter to be self!
        self.x = x
        self.y = y
```

In [ ]:
```
v1 = VectorV1(1.0,2.0)
```

In [ ]:
```
dir(v1)
```

In [ ]:
```
print(v1.dim)
print(v1.x)
print(v1.y)
```

Btw, there is nothing mysterious about the `__init__` : you can just assume it is a function (method) stored in v1, and you can always call it if you like!

When you write `v1.__init__()` , you can equivalently think that you are calling a function with "ugly function name" `__init__` , and the parameter is `v1` (self), i.e. you are writing `__init__(v1)` . It is just a function updating the attributes of instance objects!

In [ ]:
```
print(v1.x)
print(id(v1))
v1.__init__()
print(v1.x)
print(id(v1))
```

Another secret uncovered: `v1` is just a mutable object, and the "function" `__init__( )` just change `v1` in place!

Now we move on to update our vector class by defining more functions. Since you may not like ugly names here with dunder, let's just begin with normal function names.

In [ ]:
```
class VectorV2:
    '''define the vector'''  # this is the document string
    dim = 2   # this is the attribute

    def __init__(self, x=0.0, y=0.0):  # any method in Class requires the first parameter to be self!
        '''initialize the vector by providing x and y coordinate'''
        self.x = x
        self.y = y

    def norm(self):
        '''calculate the norm of vector'''
        return math.sqrt(self.x**2+self.y**2)

    def vector_sum(self, other):
        '''calculate the vector sum of two vectors'''
        return VectorV2(self.x + other.x, self.y + other.y)

    def show_coordinate(self):
        '''display the coordinates of the vector'''
        return 'Vector(%r, %r)' % (self.x, self.y)
```

In [ ]:
```
help(VectorV2)
```

In [ ]:
```
import math
v1 = VectorV2(1.0,2.0)
v2 = VectorV2(2.0,3.0)
```

In [ ]:
```
v1.norm()
```

In [ ]:
```
v3 = v1.vector_sum(v2)
v3.show_coordinate()
```

In [ ]:
```
v1+v2 # will it work?
```

In [ ]:
```
print(v3)
```

Something that we are still not satisfied:

- By typing v3 or using `print()` in the code, we cannot show its coordinates directly
- We cannot use the `+` operator to calculate the vector sum

Special (Magic) Methods

Here's the magic: by merely changing the function name, we can realize our goal!

In [ ]:
```
class VectorV3:
    '''define the vector'''  # this is the document string
    dim = 2   # this is the attribute

    def __init__(self, x=0.0, y=0.0):  # any method in Class requires the first param
eter to be self!
        '''initialize the vector by providing x and y coordinate'''
        self.x = x
        self.y = y

    def norm(self):
        '''calculate the norm of vector'''
        return math.sqrt(self.x**2+self.y**2)

    def __add__(self, other):
        '''calculate the vector sum of two vectors'''
        return VectorV3(self.x + other.x, self.y + other.y)
    def __repr__(self):
        '''display the coordinates of the vector'''
        return 'Vector(%r, %r)' % (self.x, self.y)
```

In [ ]:
```
help(VectorV3)
```

In [ ]:
```
v1 = VectorV3(1.0,2.0)
v2 = VectorV3(2.0,3.0)
```

In [ ]:
```
v3 = v1.__add__(v2)
v3.__repr__()
```

In [ ]:
```
v1+v2
```

In [ ]:
```
v3
```

Special methods are just like VIP admissions to take full use of the built-in operators in Python. With other special methods, you can even get elements by index `v3[0]`, or iterate through the object you created. For more advanced usage, you can see here (https://rszalski.github.io/magicmethods/).

Inheritance

Now we want to add another scalar production method to Vector, but we're tired of rewriting all the other methods. A good way is to create new Class VectorV4 (Child Class) by inheriting from VectorV3 (Parent Class) that we have already defined.

In [ ]:
```
class VectorV4(VectorV3): # Note the class VectorV3 in parentheses here
    '''define the vector'''  # this is the document string
    def __mul__(self, scalar):
        '''calculate the scalar product'''
        return VectorV4(self.x * scalar, self.y * scalar)
```

In [ ]:

```
help(VectorV4)
```

In [ ]:

```
v1 = VectorV4(1.0,2.0)
v2 = VectorV4(2.0,3.0)
```

In [ ]:

```
v1+v2
```

In [ ]:

```
v1*2
```