Software Development

Univ.AI

Putting re-usable code in modules

Earlier we used a module to compute the area of a circle. or make a sum of squares

```
from math import sqrt, pi # importing python builtins
def circle_area(radius):
    area = pi*radius*radius # calculate area
    return area # return the area
circle(1) # returns pi
hypot = lambda x, y : sqrt(x*x + y*y)
hypot(3, 4) # returns 5
```

While our `Vector` code still has a lot of problems, which we will deal with later, lets make a module from our vector: we save our code into a file `vector.py` in the same folder. Now we can use it in our notebook:

```
import vector
v1 = vector.Vector([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
v1 # Vector([1, 2, 3, 4, 5, 6, ...])
v1 + [1, 2, 3] # Vector([2, 4, 6])
```

`vector.py`

```
import reprlib
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 __add__(self, other_vector):
        try:
            sumlist = []
            for i, _ in enumerate(other_vector):
                sumlist.append(self._storage[i] + other_vector[i])
            return Vector(sumlist)
        except TypeError:
            return NotImplemented

    def __radd__(self, other_vector):
        # turn other + self around
        return self + other_vector

    def __mul__(self, scalar):
        return Vector([item*scalar for item in self._storage])

    def __rmul__(self, scalar):
        return self*scalar

    def __repr__(self):
        components = reprlib.repr(self._storage)
        return f"Vector({components})"
```

Testing

- As we make code changes, we want to be sure that our code is not introducing errors into the computations on vectors
- So we take all the examples we have been collecting and put them into a test area. Now we'll make sure these examples *ran the way they ran before* when we make *any code changes*.
- We'll start by introducing the simplest way to do this: *doctests*. This puts tests into the *documentation strings* of a module.
- These tests will then serve as examples of the usage of our code..and examples are probably the only documentation people read...

Documentation AND Testing

- Documentation strings are great for documenting modules, classes, and functions.
- It does not matter what you write in there as long as you provide a good description. Writing examples in the format shown on the right will ensure that the examples turn into tests.
- If you like a more formal and verbose style, use the NumPy Conventions.
- The `if __name__ == "__main__":` section at the bottom will be run on `python vector2.py -v`. (This can be used to make modules into programs). The code there will run the doctests and say `Test Passed`.

```
"""
The Vector class lets us do common operations such as
addition, scalar multiplications and dot products.

>>> v1 = Vector([4, 2, 7])
>>> v2 = Vector([1, -1, 3])
>>> v1+v2
Vector([5, 1, 10])
"""
import reprlib
class Vector:

    def __init__(self, lst):
        """
        Create a Vector from a sequence.
        """
        self._storage = lst

    def __len__(self):
        """
        Delegate length to length of storage.
        >>> v = Vector(range(10))
        >>> len(v)
        10
        """
        return len(self._storage)
...

if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

Test Driven Development

(a) Usually you will *write some code*, test it. But its also good to **(b)** *write some tests for code not yet written* (`vector3.py`) **(c)** Some tests will pass, and **(d)** those that fail will help you write new code (`vector4.py`).

We want our `Vector` class to behave like a true vector in many dimensions: adding a smaller vector (essentially a vector confined to a subspace of a larger space) should not truncate the dimension. (we show this here).

```
"""
>>> v1 = Vector([4, 2, 7])
>>> v2 = Vector([1, -1, 3])
...
>>> v1 + range(2)
Vector([4, 3, 7])
>>> range(2) + v1
Vector([4, 3, 7])
"""
```

If we want to add new features, such as a dot product `@`, we should write up what we expect from the dot product, so that we know out implementation is good.

```
"""
>>> v1 = Vector([4, 2, 7])
>>> v2 = Vector([1, -1, 3])
...
>>> v1@v2
23
>>> v2@v1
23
>>> v1 @ [-1, -1, 3]
15
>>> [-1, -1, 3] @ v1
15
"""
```

First fail, and then write code

```
import vector3
doctest.testmod(vector3, verbose=True)
```

```
**********************************
1 items had failures:
   6 of  15 in vector3
17 tests in 10 items.
11 passed and 6 failed.
***Test Failed*** 6 failures.
```

Tests for lower dimensional lists and dot products fail. Lets rectify this.

We define a function to pad vectors (with its own doctests), and use it:

```
def pad_vectors(left, right):
    """
    pad sequence left or right with zeros to make
    both the length of the longest sequence

    >>> pad_vectors(range(2), range(5,10))
    ([0, 1, 0, 0, 0], [5, 6, 7, 8, 9])
    >>> pad_vectors([1, 2, 3], range(10))
    ([1, 2, 3, 0, 0, 0, 0, 0, 0, 0], [0, 1, 2,
            3, 4, 5, 6, 7, 8, 9])
    """
    ...

def __add__(self, other_vector):
        """
        Adding 2 vectors, pads to longest length
        """
        try:
            left, right = pad_vectors(self, other_vector)
            return Vector([a + b for a, b in zip(left, right)])
        except TypeError:
            return NotImplemented

def __matmul__(self, other_vector):
        try:
            left, right = pad_vectors(self, other_vector)
            return sum([a * b for a, b in zip(left, right)])
        except TypeError:
            return NotImplemented
```

Writing Software

Now we test again: `!python vector4.py -v` (in a notebook cell or without the leading bang in the terminal):

```
19 tests in 13 items.
19 passed and 0 failed.
Test passed.
```

And voila, we have two new features. Tested. The immportant takeaways here:

- document with examples
- test, test, test
- maybe even do test driven development (TDD): write tests first and then fill in the code and make the tests work