Vectors

and their

Operators

Univ.AI

From Dunder Methods to Operator Overloading

- operations such as `+`, `-`, `*`, `/` are defined for built-in types such as numbers.
- we often want to define these for other types that we have in python: for lists and strings, etc this is done for you
- How is this done? *Dunder Methods*! For `+` you define `__add__`, for `*`, `__mul__` and `__rmul__`. You can do this for your own classes, like `Vector`!
- The Python Data Model lists all the overloadable operators.

```
1 + 1 # 2

'abc' +'def' # 'abcdef'

[1, 2, 3] + [4, 5, 6] # [1, 2, 3, 4, 5, 6]
```

For `Vector` we'd like:

```
v1, v2 = Vector([4, 2]), Vector([1, -1])
v1+v2  = Vector([5, 1])
v2+v1  = Vector([5, 1])

λ = 3.0
v3 = v1*λ
v3 # Vector([12.0, 6.0])

v4 = λ*v1
v4 # Vector([12.0, 6.0])
```

Vector addition

What we want:

```
v1 = Vector([4, 2])
v2 = Vector([1, -1])
v1+v2 # Vector([5, 1])

v1 = Vector([4, 2, 7])
v2 = Vector([1, -1, 3])
v1+v2 # Vector([5, 1, 10])
```

To implement addition,

```
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):
        sumlist = []
        for i, _ in enumerate(other_vector):
            sumlist.append(self.storage[i] +\
                 other_vector[i])
        return Vector(sumlist)

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

Our addition implementation

```
v1 = Vector([4, 2, 7])
v2 = Vector([1, -1, 3])

v1 + v2 # Vector([5, 1, 10])
v2 + v1 # Vector([5, 1, 10])

v1.__add__(v2) # Vector([5, 1, 10])

v1 + [-1, -1, 3] # Vector([3, 1, 10])

v1 + range(3) # Vector([4, 3, 9])

v1 + range(2) # Vector([4, 3])

[-1, -1, 3] + v1 # TypeError: can only
# concatenate list (not "Vector") to list

v1 + 5 # TypeError: 'int' object is not iterable
```

- What happens when you add v2 to v1? `v1.__add__(v2)`. The other way? `v2.__add__(v2)` Thus as long as v1 and v2 are vectors, addition is *commutative*, ie, `v1+v2 = v2+v1`.
- But, since our implementation iterates over anything iterable to do the addition, we can add lists, tuples, and ranges of the same size
- If we add a smaller sized list or range, our Vector's dimensionality gets cut. This is probably NOT what we want. Adding a scalar to the Vector causes issues
- Adding a list to a vector causes issues. How should we fix this?

Vector Addition (contd)

```
class Vector:
    ...
    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


v1 = Vector([4, 2, 7])
[-1, -1, 3] + v1 # Vector([3, 1, 10])
v1 + 5 # TypeError: unsupported operand
# type(s) for +: 'Vector' and 'int'
```

- In the Python Data Model, dunder methods starting with `__r` need to be implemented to figure when the new class is on the right side of the operator. Here we define `__radd__`, which works by putting on the left side
- We also see an example of Python's error handling, using `try` and `catch`. If we get a type error (as in adding an integer) we return `NotImplemented` which allows Python to try right addition, in case the other type implements left addition with something like a vector (not true for integers)

Scalar Multiplication

What we get:

```
v1 = Vector([4, 2])
λ = 3.0
v2 = v1*λ
v2 # Vector([12.0, 6.0])

v3 = λ*v1
v3 # Vector([12.0, 6.0])
```

Multiplication must be *commutative*, that is, putting the vector first or the scalar first should not make a difference.

```
class Vector:
    ...
    # we add these methods
    def __mul__(self, scalar):
        "Handles right mult: vector*scalar"
        return Vector([item*scalar for item in self.storage])

    def __rmul__(self, scalar):
        "Handles left mult: scalar*vector"
        return self*scalar # reverse the order
```