# Object-Oriented Programming II

start with a class
- assume all variables are private
- get()


Sareh Taebi

FLORIDA ATLANTIC UNIVERSITY

# Operator Overloading

- Python's built-in classes provide natural semantics for many operators.
- For example, the syntax a + b invokes addition for numeric types, yet concatenation for sequence types.
- When defining a new class, we must consider whether a syntax like a + b should be defined when a or b is an instance of that class.

can add a definition in our class to add objects

# Overloaded Operations

- can define in a class
- overloading the fundamental operators in python

- Table 2.1, page 75 has the complete list

how to add a new definition

| Common Syntax | Special Method Form | |
|---|---|---|
| a + b | a.__add__(b); | alternatively b.__radd__(a) |
| a − b | a.__sub__(b); | alternatively b.__rsub__(a) |
| a * b | a.__mul__(b); | alternatively b.__rmul__(a) |
| a / b | a.__truediv__(b); | alternatively b.__rtruediv__(a) |
| a // b | a.__floordiv__(b); | alternatively b.__rfloordiv__(a) |
| a % b | a.__mod__(b); | alternatively b.__rmod__(a) |
| a ** b | a.__pow__(b); | alternatively b.__rpow__(a) |
| a << b | a.__lshift__(b); | alternatively b.__rlshift__(a) |
| a >> b | a.__rshift__(b); | alternatively b.__rrshift__(a) |
| a & b | a.__and__(b); | alternatively b.__rand__(a) |
| a ^ b | a.__xor__(b); | alternatively b.__rxor__(a) |
| a \| b | a.__or__(b); | alternatively b.__ror__(a) |

b right and a
- alternative to add
- consider right side

# Vector Implementation

<0, 1, 2>
+
<1, 2, 3>
=
<1, 3, 5>

Goal: create a vector
- class name: Vector
- constructs a vector of whatever size we want

- It's acting differently than a list

```
                v = Vector(5) size = 5   # construct five-dimensional <0, 0, 0, 0, 0>
OL setitem -->  v[1] = 23                # <0, 23, 0, 0, 0> (based on use of __setitem__)
                v[−1] = 45               # <0, 23, 0, 0, 45> (also via __setitem__)
OL getitem -->  print(v[4])              # print 45 (via __getitem__)
                u = v + v                # <0, 46, 0, 0, 90> (via __add__)
                print(u)                 # print <0, 46, 0, 0, 90>
                total = 0
                for entry in v:          # implicit iteration via __len__ and __getitem__
                  total += entry
```

should be able to access each index by just calling the index
- fully functional without calling get/set function
- can already do all of this with sets
--- add two sets- concatenate sets
----- not adding corresponding indices

a + b = add elements of a to b
- polymorphism
- assumes a is a vector, if not it will crash

len
- int, float, dict, set, tuple
- does not meant to work on vector b/c new class
--- needs to be added
--- meant to be used for built in

# Vector Class

use this class in assignment

OL = over loading

memorize
- initiator
- get item
- len

an instance of list

```
class Vector:
  """Represent a vector in a multidimensional space."""
```

define space with constructor

```
  def __init__(self, d):
    """Create d-dimensional vector of zeros."""
    self._coords = [0] * d
```

[0] is a set, using set operators

```
  def __len__(self):
    """Return the dimension of the vector."""
    return len(self._coords)
```

return # of dimensions

```
  def __getitem__(self, j):
    """Return jth coordinate of vector."""
    return self._coords[j]
```

getitem = OL

returns value of set

```
  def __setitem__(self, j, val):
    """Set jth coordinate of vector to given value."""
    self._coords[j] = val
```

need index & new value

```
  def __add__(self, other):
    """Return sum of two vectors."""
    if len(self) != len(other):          # relies on __len__ method
      raise ValueError('dimensions must agree')
    result = Vector(len(self))           # start with vector of zeros
    for j in range(len(self)):
      result[j] = self[j] + other[j]
    return result
```

add two corresponding indices

self = left
raise exception, must be same length
or write loop & add corresponding indices

```
  def __eq__(self, other):
    """Return True if vector has same coordinates as other."""
    return self._coords == other._coords
```

comparison

```
  def __ne__(self, other):
    """Return True if vector differs from other."""
    return not self == other             # rely on existing __eq__ definition
```

not equal

```
  def __str__(self):
    """Produce string representation of vector."""
    return '<' + str(self._coords)[1:−1] + '>'  # adapt list representation
```

don't have to call methods or anything just str(v)

concatenating all vector values into string
- should not have <> if its a vector

read on our own

# Iterators

- Iteration is an important concept in the design of data structures.
- An **iterator** for a collection provides one key behavior:
  - It supports a special method named `__next__` that returns the next element of the collection, if any, or raises a StopIteration exception to indicate that there are no further elements.

# Automatic Iterators

replicates range
review on our own

- Python also helps by providing an automatic iterator implementation for any class that defines both `__len__` and `__getitem__`.

 - use those & access automatic iteration

__name__
- built ins
- make class practical

```
class Range:
  """A class that mimic's the built-in range class."""

  def __init__(self, start, stop=None, step=1):
    """Initialize a Range instance.

    Semantics is similar to built-in range class.
    """
    if step == 0:
      raise ValueError('step cannot be 0')

    if stop is None:                  # special case of range(n)
      start, stop = 0, start          # should be treated as if range(0,n)

    # calculate the effective length once
    self._length = max(0, (stop - start + step - 1) // step)

    # need knowledge of start and step (but not stop) to support __getitem__
    self._start = start
    self._step = step

  def __len__(self):
    """Return number of entries in the range."""
    return self._length

  def __getitem__(self, k):
    """Return entry at index k (using standard interpretation if negative)."""
    if k < 0:
      k += len(self)                  # attempt to convert negative index

    if not 0 <= k < self._length:
      raise IndexError('index out of range')

    return self._start + k * self._step
```

default values
if not user input

# Inheritance

- A mechanism for a modular and hierarchical organization is **inheritance**.
- This allows a new class to be defined based upon an existing class as the starting point.
- The existing class is typically described as the **base class**, parent class, or superclass, while the newly defined class is known as the **subclass** or child class.
- There are two ways in which a subclass can differentiate itself from its superclass:
  - A subclass may specialize an existing behavior by providing a new implementation that overrides an existing method.
  - A subclass may also extend its superclass by providing brand new methods.