Encapsulation

- Why?
- Python *private* conventions
  - single and double underscores
- MySet Examples
  - Two *homemade* implementations of sets
    - Same interface
    - Using lists
    - Using dictionaries

Principles of OOP

- **Encapsulation**
  - Encapsulation is the mechanism of hiding of data implementation by restricting access to public methods
- **Inheritance**
  - Inheritances expresses "is a" relationship between two objects. Using proper inheritance, In derived classes we can reuse the code of existing super classes
- **Polymorphism**
  - It means one name many forms. Details of what a method does will depend on the object to which it is applied.
- Also
  - **Instantiation**
  - **Abstraction**
  - **Modularity**

Encapsulation Principles

- Hide implementation details within the object
- Object is accessed through defined interface
  - public methods
  - public variables / data

- In other languages
  - That which should not be accessed (methods, data) should not be accessible
  - Java has **private** methods and attributes to enforce this
- Python has no real concept of **private**
  - Except by convention

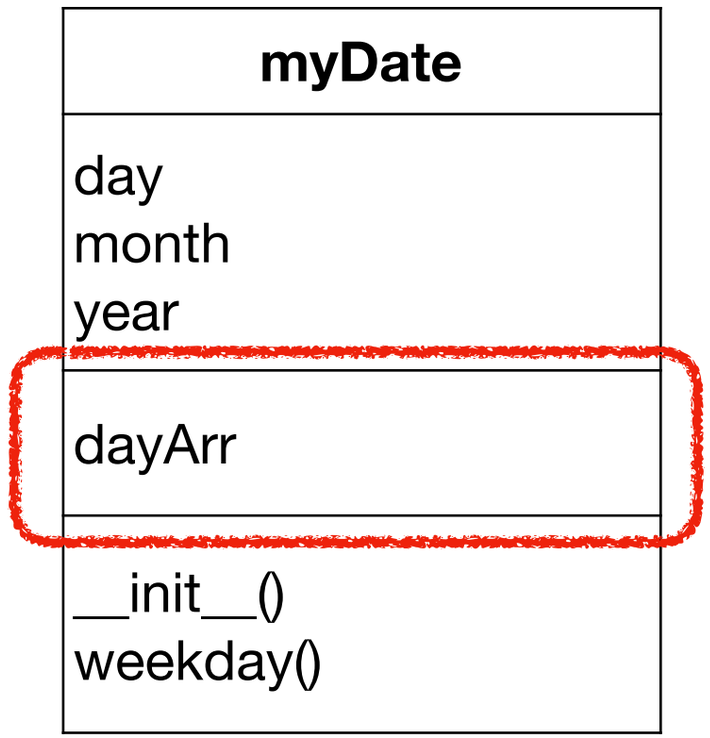
Aside on Java

- Standard OOP (e.g. Java)
  - Classes will have private variables
    - Not externally accessible
    - Accessible through ‘getters’ and ‘setters’ if necessary

```
public class Dog extends Animal {

    private int numberOfLegs;
    private boolean hasOwner;

    public Dog() {
        numberOfLegs = 4;
        hasOwner = false;
    }

    private void bark() {
        System.out.println("Woof!");
    }

    public void move() {
        System.out.println("Running");
    }

}
```

What could/should be Private?

- Imagine a myDate class that holds dates as day/month/year
- It has a method weekday() that returns the weekday
  - `nyd = myDate(1,1,2000)`
  - `nyd.weekday()`
  - `–> Saturday`
- This class has attributes day, month and year that can be **public**
  - `nyd.day`
  - `–> 1`
- It uses a list of day names
  `["Monday","Tuesday",…"Sunday"]`
  - This should be **private**



Underscore conventions

- single_leading_underscore:
  - weak "internal use" indicator
  - i.e. should not be accessed outside the class
  - private by convention
- double_leading_underscore:
  - name ***mangling*** supports privacy
- double_leading_and_trailing_underscore:
  - e.g. __init__, __file__, __import__, __eq__,__lt__
  - magic methods
  - don’t invent new ones

Underscore Conventions

```
class EncapDemo():
    def __init__(self):
        self.v1 = 11
        self._v2 = 22
        self.__v3 = 33

    def incBy_v3(self, x):
        return x + self.__v3
```

```
ed = EncapDemo()
In [12]:
ed.v1
Out[12]:
11
In [13]:
ed._v2
Out[13]:
22
In [14]:
ed.__v3
AttributeError …
```

Single underscore
Not hidden

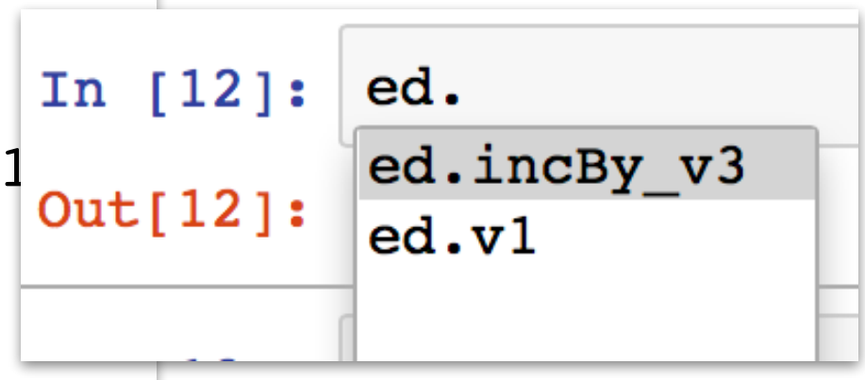
Double underscore
Hidden

```
ed._EncapDemo__v3
Out[15]:
33
In [16]:
ed.__dict__
Out[16]:
{'_EncapDemo__v3': 33, '_v2': 22, 'v1': 1
In [17]:
ed.incBy_v3(44)
Out[17]:
77
```

Name mangling

Accessible by class methods

Tab completion



UnderscoreConv notebook

MySet class

- MySet class with methods
  - constructor
  - clear
  - add
  - member
  - remove
  - union
  - intersection
  - print
  - mem_list (return the members as a list)

Sets
Like a list except:
No duplicates
No order

- Two implementations
  - One with lists
  - One using a dictionary
  - Because of encapsulation these are interchangeable.

MySet: V1 using a list

```
class MySet:
    def __init__(self, members):
        self._members = []
        for cand in members:
            self.add(cand)

    def clear(self):
        self._members = []

    def add(self,cand):
        if not cand in self._members:
            self._members.append(cand)

    def member(self, mem):
        for el in self._members:
            if el == mem: return True
        return False
```

```
    def remove(self,mem):
        self._members.remove(mem)

    def union(self,s):
        su = MySet(self._members)
        for el in s._members:
            su.add(el)
        return su

    def intersection(self,s):
        si = MySet([])
        for el in self._members:
            if el in s._members:
                si.add(el)
        return si
```

```
def print(self):
    mems = self._members
    print('V1{',end='')
    for i in range(len(mems)):
        print(mems[i],end='')
        if i != len(mems)-1:
            print(', ',end='')
    print('}')

def mem_list(self):
    return self._members
```

MySet in Action

```
s4 = MySet([1,2,3,3,2,4])
s4.print()

V1{1, 2, 3, 4}
In [8]:
print(s4.member(4))
print(s4.member(7))

True
False
In [9]:
s4.clear()
s4.print()

V1{}
In [10]:
s4.add(9)
s4.add(8)
s4.print()

V1{9, 8}
```

```
In [11]:
s1 = MySet(['a','b','c','d'])
s2 = MySet(['x','c','y','b'])
s1us2 = s1.union(s2)
s1is2 = s1.intersection(s2)
In [12]:
s1.print()
s2.print()
s1us2.print()
s1is2.print()

V1{a, b, c, d}
V1{x, c, y, b}
V1{a, b, c, d, x, y}
V1{b, c}
```

MySet notebook

MySetV2

- A new implementation of MySet using a dictionary
  - key:value pairs - value is not used: “DVal” is a dummy value

MySetV1

```
class MySet:
    def __init__(self, members):
        self._members = []
        for cand in members:
            self.add(cand)
```

stored internally as a list

Why?
Why change from a set to a dictionary?

MySetV2

stored internally as a dictionary

```
class MySet:
    def __init__(self, members):
        self._members = {}
        for cand in members:
            self.add(cand)
```

MySet V1 & V2 Compared

MySetV1

```
def clear(self):
        self._members = []

def add(self,cand):
    if not cand in self._members:
            self._members.append(cand)

def member(self, mem):
    for el in self._members:
        if el == mem: return True
    return False

def remove(self,mem):
    self._members.remove(mem)

def union(self,s):
    su = MySet(self._members)
    for el in s._members:
        su.add(el)
    return su

def intersection(self,s):
    si = MySet([])
    for el in self._members:
        if el in s._members:
            si.add(el)
    return si
```

MySetV2

```
def clear(self):
    self._members = {}

def add(self,cand):
    if not cand in self._members:
            self._members[cand]="DVal"

def member(self, mem):
    if mem in self._members: return True
    return False

def remove(self,mem):
    del self._members[mem]

def union(self,s):
    su = MySet(self._members.keys())
    for el in s._members.keys():
        su.add(el)
    return su

def intersection(self,s):
    si = MySet([])
    for el in self._members.keys():
        if el in s._members.keys():
            si.add(el)
    return si
```

Exercise

- In both V1 and V2 the data (._members) is weakly private
- Change this to a double underscore to see what happens

- Can you still access the data directly?
  - i.e. without using the class methods