Object Oriented Programming Concepts

- **Encapsulation:** Objects know stuff and do stuff, combining data and operations

- **Polymorphism:** Many forms

- **Inheritance:** New class can be defined to borrow behavior from another class
Encapsulation

- The internal representation of an object cannot be seen from outside of the object's definition. Access to this data is typically only achieved through special methods.

- Closely related keywords: Abstraction, Hiding
Encapsulation example

class CSet:
    def __init__(self):
        self.elements = []

    def addElement(self, element):
        if element not in self.elements:
            self.elements.append(element)

    def elementExists(self, element):
        return element in self.elements

    def printElements(self):
        for element in self.elements:
            print(element)
Encapsulation example

class CSet:
    def __init__(self):
        self.elements = []

    def addElement(self, element):
        if element not in self.elements:
            self.elements.append(element)

    def elementExists(self, element):
        return element in self.elements

    def printElements(self):
        for element in self.elements:
            print(element)
Encapsulation example

- Someone who is using our CSet class do not need to know the internal structure of it.
- One only need to know the following functions to use our CSet class:
  - addElement()
  - elementExists()
  - printElements()
Encapsulation example

• We can also change the internal structure of CSet, to make it:
  
  ⚫ Faster
  
  ⚫ More stable
  
  ⚫ Add more features
  
• But it is not going to affect the person who uses it.

• Because CSet is encapsulated!
Few notes on Python encapsulation

- Python doesn't support strong encapsulation.
- You can not hide private variables
- There's an agreement that underscores mean private elements and you should not use them. Unless you know what you're doing and you really want to.
- The reason is simply philosophy.
Polymorphism

• Many forms

• Polymorphism is the ability (in programming) to present the same interface for differing underlying forms (data types)

• For example:

  - point.draw()
  - circle.draw()
  - rectangle.draw()
Polymorphism example

```python
class Wolf(object):
    def bark(self):
        print "hooooowll"

class Dog(object):
    def bark(self):
        print "woof"

def barkforme(myAnimal):
    myAnimal.bark()

myDog = Dog()
myWolf = Wolf()
barkforme(myDog)
barkforme(myWolf)
```
Polymorphism example

```python
1 class Wolf(object):
2     def bark(self):
3         print "hooooowll"
4
5 class Dog(object):
6     def bark(self):
7         print "woof"
8
9 def barkforme(myAnimal):
10     myAnimal.bark()
```

```python
12 myDog = Dog()
13 myWolf = Wolf()
14 barkforme(myDog)
15 barkforme(myWolf)
```

Result:

```
woof
hooooowll
```
Polymorphism example

- We have the same interface for barking in both wolf and dog object.
  - myDog.bark()
  - myWolf.bark()
- barkforme() function do not need to explicitly know the type of its input, it is enough for input to have a valid bark() function
Inheritance

- Inheritance is when an object or class is based on another object or class, using the same implementation (inhertiting from a class) or specifying implementation to maintain the same behavior (realizing an interface; inheriting behavior).

- It is a mechanism for code reuse and to allow independent extensions of the original software via public classes and interfaces.
```python
class Mammal(object):
    def __init__(self, name):
        self.name = name
        self.nlegs = 4
        print 'New mammal born: ' + self.name

    def eat(self):
        print self.name + ' eating now.'

    def die(self):
        print self.name + ' died :'(

class Cat(Mammal):
    def __init__(self, name, catType):
        super(Cat, self).__init__(name)
        self.catType = catType

    def meow(self):
        print(self.name + ': meooow!' + ' ('+self.catType+')')

class Dog(Mammal):
    def __init__(self, name, dogType):
        super(Dog, self).__init__(name)
        self.dogType = dogType

    def bark(self):
        print(self.name + ': woof!' + ' ('+self.dogType+')')
```
Inheritance example

class Mammal(object):
    def __init__(self, name):
        # ...

def eat(self):
    # ...

def die(self):
    # ...

class Cat(Mammal):
    def __init__(self, name, catType):
        # ...

def meow(self):
    # ...

class Dog(Mammal):
    def __init__(self, name, dogType):
        # ...

def bark(self):
    # ...

cat = Cat('Garfield', 'Siberian')
cat.meow()
dog = Dog('Buck', 'Husky')
dog.bark()
cat.die()
Inheritance example

```python
class Mammal(object):
    def __init__(self, name):
        # ...

def eat(self):
    # ...

def die(self):
    # ...

class Cat(Mammal):
    def __init__(self, name, catType):
        # ...

def meow(self):
    # ...

class Dog(Mammal):
    def __init__(self, name, dogType):
        # ...

def bark(self):
    # ...

cat = Cat('Garfield', 'Siberian')
cat.meow()
dog = Dog('Buck', 'Husky')
dog.bark()
cat.die()
```

Result:

New mammal born: Garfield
Garfield: meooow! (Siberian)
New mammal born: Buck
Buck: woof! (Husky)
Garfield died :(

Instance variables

• Some instance variables can be part of the interface.

class Person:
    name = 'John Doe'
    def __init__(self):
        print 'A new person born'

person1 = Person()
print person1.name

person2 = Person()
person2.name = 'Sait'
print person2.name
Recursion

- Recursion is the process of repeating items in a self-similar way.
RECURSION
Here we go again

RECURSION
Here we go again

RECURSION
Here we go again

RECURSION
Here we go again

RECURSION
Here we go again

RECURSION
Here we go again

RECURSION
Here we go again

RECURSION
Here we go again
Sierpinski triangle
Factorial (Recursion)

```python
1 def factorial(n):
2     if n < 2:
3         return 1
4     else:
5         return n * factorial(n-1)
6
7 for i in range(2, 11):
8     print i, factorial(i)
```
Factorial (Recursion)

def factorial(n):
    if n < 2:
        return 1
    else:
        return n * factorial(n-1)

for i in range(2, 11):
    print(i, factorial(i))
Factorial (Recursion)

```python
def factorial(n):
    print 'factorial(' + str(n) + ') called.'
    if n < 2:
        return 1
    else:
        print 'return ' + str(n) + ' * factorial(' + str(n-1) + ')'
        return n * factorial(n-1)

factorial(10)
```
factorial(10) called.
return 10 * factorial(9)
factorial(9) called.
return 9 * factorial(8)
factorial(8) called.
return 8 * factorial(7)
factorial(7) called.
return 7 * factorial(6)
factorial(6) called.
return 6 * factorial(5)
factorial(5) called.
return 5 * factorial(4)
factorial(4) called.
return 4 * factorial(3)
factorial(3) called.
return 3 * factorial(2)
factorial(2) called.
return 2 * factorial(1)
factorial(1) called.
Factorial (Iterative)

```python
def factorial(n):
    rVal = 1
    for i in range(2, n+1):
        rVal = rVal * i
    return rVal

print(factorial(4))
```

Result: ?
Factorial (Iterative)

```python
def factorial(n):
    rVal = 1
    for i in range(2, n+1):
        rVal = rVal * i
    return rVal

print(factorial(4))
```

Result: 24
Factorial (Iterative)

```python
1 def factorial(n):
2     rVal = 1
3     for i in range(2, n+1):
4         rVal = rVal * i
5     return rVal
6
7     for i in range(2,10):
8     print str(i) + ' -> ' + str(factorial(i))
```
Factorial (Iterative)

```python
def factorial(n):
    rVal = 1
    for i in range(2, n+1):
        rVal = rVal * i
    return rVal

for i in range(2,10):
    print str(i) + ' -> ' + str(factorial(i))
```

Result:

```
2 -> 2
3 -> 6
4 -> 24
5 -> 120
6 -> 720
7 -> 5040
8 -> 40320
9 -> 362880
```
Tree traversal

- Pre-order
- In-order
- Post-order
class Node:
    def __init__(self, value, parent):
        self.value = value
        self.parent = parent
        self.leftChild = None
        self.rightChild = None

    def isRoot(self):
        return self.parent == None

    def isLeaf(self):
        return self.leftChild == None and self.rightChild == None
Building the tree

```python
15 nodeF = Node('F', None)
16 nodeB = Node('B', nodeF)
17 nodeF.leftChild = nodeB
18 nodeG = Node('G', nodeF)
19 nodeF.rightChild = nodeG
20 nodeA = Node('A', nodeB)
21 nodeB.leftChild = nodeA
22 nodeD = Node('D', nodeB)
23 nodeB.rightChild = nodeD
24 nodeC = Node('C', nodeD)
25 nodeD.leftChild = nodeC
26 nodeE = Node('E', nodeD)
27 nodeD.rightChild = nodeE
28 nodeI = Node('I', nodeG)
29 nodeG.rightChild = nodeI
30 nodeH = Node('H', nodeI)
31 nodeI.leftChild = nodeH
```

Diagram:
```
  F
 / \    
B   G
   / \   
  A   D   I
    / \    
   C   E   H
```
Pre-order traversal
Pre-order traversal

```python
42  def preorder(node):
43      if node != None:
44          print node.value
45          preorder(node.leftChild)
46          preorder(node.rightChild)
```
Pre-order traversal
In-order traversal
In-order traversal

```python
48 def inorder(node):
49     if node != None:
50         inorder(node.leftChild)
51         print node.value
52         inorder(node.rightChild)
```
In-order traversal
Post-order traversal
def postorder(node):
    if node != None:
        postorder(node.leftChild)
        postorder(node.rightChild)
        print node.value
Post-order traversal
Fibonacci example

- Fibonacci sequence:
  
  1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, ...

- \( F(1) = 1, F(2) = 1 \)

- \( F(n) = F(n-1) + F(n-2) \quad \text{if } n > 2 \)
Fibonacci example (Recursion)

```python
def fibonacci(n):
    if n == 0:
        return 1
    if n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)

for i in range(10):
    print(fibonacci(i))
```
Fibonacci example (Recursion)

```python
1   def fibonacci(n):
2       if n == 0:
3           return 1
4       if n == 1:
5           return 1
6       else:
7           return fibonacci(n-1) + fibonacci(n-2)
8
9   for i in range(10):
10      print(fibonacci(i))
```

Fibonacci example (Recursion)

• This implementation is naive. (and slow)

• Calculating the same values again.
Fibonacci example (Recursion)

```
fib(7)
  /    /
/fib(6)  /  
|      /    /    
/fib(5) /  
|      |  /  
/fib(4) /  
|      |  /  
/fib(3) /  
|      |  /  
/fib(2) /  
|      |  /  
/fib(1) /  
```

Fibonacci example (Recursion)

```python
1 def fibonacci(n):
2     print 'fibonacci('+str(n)+') called.'
3     if n == 0:
4         return 1
5     if n == 1:
6         return 1
7     else:
8         return fibonacci(n-1) + fibonacci(n-2)
9
10 fibonacci(10)
```
fibbonacci(10)

Even more ...
Memoization is not typo.

It is a technique that stores some results of expensive function calls that might be needed again.

It makes programs faster.
def fibonacci(n, memo =\{0:0, 1:1\}):
    if n in memo:
        return memo[n]
    else:
        print 'fibonacci\('+str(n)+\') called.'
        memo[n] = fibonacci(n-1, memo) + fibonacci(n-2, memo)
        return memo[n]

print fibonacci(10)
Fibonacci (Memoization)

```python
fibonacci(10) called.
fibonacci(9) called.
fibonacci(8) called.
fibonacci(7) called.
fibonacci(6) called.
fibonacci(5) called.
fibonacci(4) called.
fibonacci(3) called.
fibonacci(2) called.

55
```
Let us recall the iterative way of Fibonacci again to see the difference between recursive way.
Recall: Fibonacci example
(Iterative)

```python
def fibonacci(n):
    fiboArray = [1,1]
    i = 2
    while i <= n:
        fiboArray.append( fiboArray[i-1] + fiboArray[i-2] )
        i += 1
    return fiboArray[n]

for i in range(10):
    print fibonacci(i)
```
Recall: Fibonacci example (Iterative)

```
1 def fibonacci(n):
2     fiboArray = [1,1]
3     i = 2
4     while i <= n:
5         fiboArray.append( fiboArray[i-1] + fiboArray[i-2] )
6         i += 1
7     return fiboArray[n]
8
9     for i in range(10):
10    print fibonacci(i)
```
Recall: Fibonacci example (Iterative)
Questions?

Thanks!