CS177 Python Programming

Recitation 14
Table of Contents

- Process of Object Oriented Design (OOD)
- Encapsulation
- Inheritance
- Polymorphism
The essence of OOD is describing a system in terms of magical black boxes and their interfaces

class Vehicle:
    def __init__(self, wheels, miles, make, model, year):
        self.wheels = wheels
        self.miles = miles
        self.make = make
        self.model = model
        self.year = year
    def set_mile(self, mile):
        self.mile = mile
    def print_vehicle(self):
        print (self.wheels, self.miles, self.make, self.model, self.year)

>>>v = Vehicle (4, 10000, 'Nissan', 'Altima', 2016)
>>>v.set_mile(20000)
>>>v.print_vehicle()
(4, 20000, 'Nissan', 'Altima', 2016)
Classes are the inside magic and Objects are the black boxes

```python
class Vehicle:
    def __init__(self, wheels, miles, make, model, year):
        self.wheels = wheels
        self.miles = miles
        self.make = make
        self.model = model
        self.year = year
    def set_mile(self, mile):
        self.mile = mile
    def print_vehicle(self):
        print (self.wheels, self.miles, self.make, self.model, self.year)

>>> v = Vehicle (4, 10000, 'Nissan', 'Altima', 2016)
>>> v.set_mile(20000)
>>> v.print_vehicle()
(4, 20000, 'Nissan', 'Altima', 2016)
```

Inside magic!!!

Object is the black box!!!
Once a class definition is written, ignore how the class works and rely on the external interface, its methods

```python
def createGameCanvas():
    win = GraphWin("MyWindow", 500, 500)
    win.setBackground("brown")
    return win

def drawShapes(win):
    sq = Rectangle(Point(100, 460), Point(140, 500))
    sq.draw(win)
    sq.setFill("red")
    cir = Circle(Point(250, 475), 25)
    cir.draw(win)
    cir.setFill("blue")
    return [sq, cir]
```

You were able to draw a circle without having to know all the nitty-gritty details encapsulated in class definitions for GraphWin and Circle.

```python
>>> win = createGameCanvas()
>>> shapesList = drawShapes(win)
```
Breaking a large problem into a set of cooperating classes reduces the complexity

Consider the drawing of Point and Circle of your Project 2

class Point:
    def __init__(x, y):
        self.x = x
        self.y = y

class Circle:
    def __init__(center, radius):
        self.center = center
        self.radius = radius

class Rectangle:
    def __init__(point1, point2):
        self.point1 = point1
        self.point2 = point2
Breaking a large problem into a set of cooperating classes reduces the complexity

class Point:
    def __init__(x, y):
        self.x = x
        self.y = y

class Circle:
    def __init__(center, radius):
        self.center = center
        self.radius = radius

class Rectangle:
    def __init__(point1, point2):
        self.point1 = point1
        self.point2 = point2

>>> sq = Rectangle(Point(100, 460),
                   Point(140, 500))
>>> sq.draw(win)
>>> sq.setFill("red")
>>> cir = Circle(Point(250, 475), 25)
>>> cir.draw(win)
>>> cir.setFill("blue")
Encapsulation is to **bundle/package the data with a set of operations (related functions)** on that data. Ideally, the data inside an object can only be manipulated by calling the object's functions.

```python
class BankAccount:
    def __init__(self, initial_balance):
        # create an account with the given balance
        self.balance = initial_balance
        self.fees = 0
    def deposit(self, amount):
        # deposit the amount to the account
        self.balance += amount
        return self.balance
    def withdraw(self, amount):
        self.balance -= amount
        return self.balance
    def get_balance(self):
        # returns the current balance in the account
        return self.balance
    def get_fees(self):
        # return the total fees deducted from the account
        return self.fees
```

It is like a black box!!!
We only need to worry about what objects can do (via the functions provided) with the member data, and not about how they are implemented.
Encapsulation is to bundle/package the data with a set of operations (related functions) on that data.

```python
class BankAccount:
    def __init__(self, initial_balance):
        # create an account with the given balance
        self.balance = initial_balance
        self.fees = 0
    def deposit(self, amount):
        # deposit the amount to the account
        self.balance += amount
        return self.balance
    def withdraw(self, amount):
        if (self.balance - amount < 0):
            # overdraw
            self.balance -= amount + 5  # $5 penalty
            self.penalty += 5
        else:
            self.balance -= amount
        return self.balance
    def get_balance(self):
        # returns the current balance in the account
        return self.balance
    def get_fees(self):
        # return the total fees deducted from the account
        return self.fees

>>> account1 = BankAccount(1000)
>>> account2 = BankAccount(2000)
>>> print (account1.get_balance(), account2.get_balance())
(1000, 2000)
>>> account1.deposit(500)
>>> print (account1.get_balance())
1500
```

account1 and account2 objects bundle the member data (balance, penalty) and related functions together. They do not need to worry about the implementation details.
The implementation can change, but as long as the interface is preserved, the object will not break

class BankAccount:
    def __init__(self, initial_balance):
        # create an account with the given balance
        self.balance = initial_balance
        self.fees = 0
    def deposit(self, amount):
        # deposit the amount to the account
        self.balance += amount
        return self.balance
    def withdraw(self, amount):
        if (self.balance - amount < 0):
            # overdraft
            self.balance -= amount + 5  # $5 penalty
            self.fees += 5
        else:
            self.balance -= amount
        return self.balance
    def get_balance(self):
        # returns the current balance in the account
        return self.balance
    def get_fees(self):
        # return the total fees deducted from the account
        return self.fees

>>> account1 = BankAccount(1000)
>>> account2 = BankAccount(2000)
>>> print (account1.get_balance(), account2.get_balance())
(1000, 2000)
>>> account1.deposit(500)

>>> print (account1.get_balance())
1500

>>> account1.withdraw(2000)

>>> print (account1.get_balance())
-505

>>> account1.get_fees(2000)
5
Inheritance is used to define a new class which can borrow behavior from another class.

We are interested to define Checking Account and Savings Account which borrow the behavior of BankAccount as shown earlier.

BankAccount:
  SavingsAccount: add interest (we'll assume 3%) on every deposit.
  CheckingAccount: charges $3 for every withdrawal.
New class **CheckingAccount** can be created inheriting from **BankAccount**

class CheckingAccount(BankAccount):
    def __init__(self, initial_balance):
        BankAccount.init(self, initial_balance)
        self.fee = 3

Super class/parent class

Calling super class’s init method
New class **CheckingAccount** can be created inheriting from **BankAccount**:

```python
class CheckingAccount(BankAccount):
    def __init__(self, initial_balance):
        BankAccount.init(self, initial_balance)
        self.fee = 3
```

**Inherited data members from** **BankAccount**:
- balance

**Inherited methods from** **BankAccount**:
- deposit (self, amount)
- withdraw (self, amount)
- get_balance (self)
- get_fees (self)
New class **CheckingAccount** can be created inheriting from **BankAccount**

```python
class CheckingAccount(BankAccount):
    def __init__(self, initial_balance):
        BankAccount.init(self, initial_balance)
        self.fee = 3

# create a checking account
>>> account1 = CheckingAccount(1000)
>>> print (account1.get_balance())
1000
>>> account1.withdraw(100)
>>> print (account1.get_balance())
500
```

Super class/parent class

Calling super class’s init method

Account1 is calling the inherited methods
New class `CheckingAccount` can be created inheriting from `BankAccount`:

```python
class CheckingAccount(BankAccount):
    def __init__(self, initial_balance):
        BankAccount.init(self, initial_balance)
        self.fee = 3
```

```python
# create a checking account
>>> account1 = CheckingAccount(1000)
>>> print (account1.get_balance())
1000
>>> account1.withdraw(100)
>>> print (account1.get_balance())
500
```

However, `CheckingAccount` has different policy ($3 charge) for withdrawal than the `BankAccount`!
Subclass methods can **override** superclass’s methods

```python
class CheckingAccount(BankAccount):
    
def __init__(self, initial_balance):
        BankAccount.init(self, initial_balance)
        self.fee = 3

    
def withdraw(self, amount):
        BankAccount.withdraw(self, amount + self.fee)
        # charge $3 for every withdrawal
        return self.balance
```

Super class/parent class

Calling super class’s init method

CheckingAccount.withdraw overrides
BankAccount.withdraw method
Subclass methods can **override** superclass’s methods

```python
class CheckingAccount(BankAccount):
    def __init__(self, initial_balance):
        BankAccount.init(self, initial_balance)
        self.fee = 3
    def withdraw(self, amount):
        BankAccount.withdraw( self, amount + self.fee)
        # charge $3 for every withdrawal
        return self.balance
# create a checking account
>>> account1 = CheckingAccount(1000)
>>> account1.deposit(100)
>>> print (account1.get_balance())
1100
>>> account1.withdraw(500)
>>> print (account1.get_balance())
597
```
New class **SavingsAccount** can be created by inheriting from **BankAccount**

class SavingsAccount(BankAccount):
    def __init__(self, initial_balance):
        BankAccount.init(self, initial_balance)

Inherited data members from **BankAccount**:
- balance
- fees

Inherited methods from **BankAccount**:
- deposit (self, amount)
- withdraw (self, amount)
- get_balance (self)
- get_fees (self)
New class **SavingsAccount** can be created by inheriting from **BankAccount**

```python
class SavingsAccount(BankAccount):
    def __init__(self, initial_balance):
        BankAccount.init(self, initial_balance)
```

**Inherited data members from BankAccount:**
- balance
- fees

**Inherited methods from BankAccount:**
- deposit (self, amount)
- withdraw (self, amount)
- get_balance (self)
- get_fees (self)

However, **SavingsAccount** has different policy for deposit *(add an interest on every deposit)* than the BankAccount!
Overloading of super class’s `deposit` method in the subclass

class SavingsAccount(BankAccount):
    def __init__(self, initial_balance):
        BankAccount.init(self, initial_balance)
    def deposit(self, amount, interest_rate):
        BankAccount.deposit(self, amount)
        self.balance = self.balance + amount * (1 + interest_rate)
        # add 3% interest on every deposit
        return self.balance

SavingsAccount.deposit has three parameters.
Overloading of super class's `deposit` method in the subclass

```python
class SavingsAccount(BankAccount):
    def __init__(self, initial_balance):
        BankAccount.init(self, initial_balance)
    def deposit(self, amount, interest_rate):
        BankAccount.deposit(self, amount)
        self.balance = self.balance + amount * (1 + interest_rate)
        # add 3% interest on every deposit
        return self.balance

Inherited methods from BankAccount:
- deposit (self, amount)
- withdraw (self, amount)
- get_balance (self)
- get_fees (self)
```

SavingsAccount.deposit has three parameters.
BankAccount.deposit has two parameters
Overloading of super class’s `deposit` method in the subclass

```python
class SavingsAccount(BankAccount):
    def __init__(self, initial_balance):
        BankAccount.init(self, initial_balance)
    def deposit(self, amount, interest_rate):
        BankAccount.deposit(self, amount)
        self.balance = self.balance + amount * (1 + interest_rate)
        # add 3% interest on every deposit
        return self.balance

Inherited methods from BankAccount:
deposit (self, amount)
withdraw (self, amount)
get_balance (self)
get_fees (self)
```

# create a savings account
```python
class1 = SavingsAccount(1000)
class1.deposit(100)
pclass1.deposit(500, 0.03)
```
Overloading of super class’s `deposit` method in the subclass

```python
class SavingsAccount(BankAccount):
    def __init__(self, initial_balance):
        BankAccount.init(self, initial_balance)
    def deposit(self, amount, interest_rate):
        BankAccount.deposit(self, amount)
        self.balance = self.balance + amount * (1 + interest_rate)
        # add 3% interest on every deposit
        return self.balance

Inherited methods from BankAccount:
- deposit (self, amount)
- withdraw (self, amount)
- get_balance (self)
- get_fees (self)
```

```python
# create a savings account
>>> account1 = SavingsAccount(1000)
>>> account1.deposit(100)
>>> print (account1.get_balance())
1100
>>> account1.deposit(500, 0.03)
>>> print (account1.get_balance())
1615
```
Polymorphism refers that what an object does in response to a message (a method call) depends on the type or class of the object.

class Shape:
    def area(self):
        return 'unknown'

class Point(Shape):
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def area(self):
        return 0

class Circle(Shape):
    def __init__(self, center, radius):
        self.center = center
        self.radius = radius
    def area(self):
        return (3.14 * radius * radius)

class Square(Shape):
    def __init__(self, length):
        self.length
    def area(self):
        return (length*length)

class Rectangle(Shape):
    def __init__(self, point1, point2):
        self.p1 = point1
        self.p2 = point2
    def area(self):
        length = p1.y-p2.y
        width = p2.x - p1.x
        return (length*width)

>>>p = Point(10, 20)
>>>c = Circle (p, 5)
>>>s = Square(6)
>>>r = rectangle(Point(10,10), Point(0,0))
>>>shapes = [p, c, s, r]
>>>for shape in shapes:
    print(shape.area())
0
78.5
36
200
Polymorphism refers to the fact that what an object does in response to a message (a method call) depends on the type or class of the object.

class Shape:
    def area(self):
        return 'unknown'

class Point(Shape):
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def area(self):
        return 0

class Circle(Shape):
    def __init__(self, center, radius):
        self.center = center
        self.radius = radius
    def area(self):
        return (3.14 * radius * radius)

class Square(Shape):
    def __init__(self, length):
        self.length
    def area(self):
        return (length*length)

class Rectangle(Shape):
    def __init__(self, point1, point2):
        self.p1 = point1
        self.p2 = point2
    def area(self):
        length = p1.y-p2.y
        width = p2.x - p1.x
        return (length*width)

>>> p = Point(10, 20)
>>> c = Circle(p, 5)
>>> s = Square(6)
>>> r = rectangle(Point(10,20), Point(0,0))
>>> shapes = [p, c, s, r]
>>> for shape in shapes:
...    print(shape.area())
0 # Point p’s area
Polymorphism refers to the fact that what an object does in response to a message (a method call) depends on the type or class of the object.

class Shape:
    def area(self):
        return 'unknown'

class Point(Shape):
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def area(self):
        return 0

class Circle(Shape):
    def __init__(center, radius):
        self.center = center
        self.radius = radius
    def area(self):
        return (3.14 * radius * radius)

class Square(Shape):
    def __init__(length):
        self.length
    def area(self):
        return (length*length)

class Rectangle(Shape):
    def __init__(point1, point2):
        self.p1 = point1
        self.p2 = point2

        def area(self):
            width = p2.x - p1.x
            length = p1.y - p2.y
            return (length*width)

>>> p = Point(10, 20)
>>> c = Circle (p, 5)
>>> s = Square(6)
>>> r = rectangle(Point(10,20), Point(0,0))
>>> shapes = [p, c, s, r]
>>> for shape in shapes:
...     print(shape.area())
0  # Point p’s area
78.5  # Circle c’s area
Polymorphism refers to the fact that what an object does in response to a message (a method call) depends on the type or class of the object.

```python
class Shape:
    def area(self):
        return 'unknown'

class Point(Shape):
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def area(self):
        return 0

class Circle(Shape):
    def __init__(self, center, radius):
        self.center = center
        self.radius = radius
    def area(self):
        return (3.14 * radius * radius)

class Square(Shape):
    def __init__(self, length):
        self.length = length
    def area(self):
        return (length*length)

class Rectangle(Shape):
    def __init__(self, point1, point2):
        self.p1 = point1
        self.p2 = point2
    def area(self):
        length = p1.y-p2.y
        width = p2.x - p1.x
        return (length*width)
```

```python
>>> p = Point(10, 20)
>>> c = Circle (p, 5)
>>> s = Square(6)
>>> r = rectangle(Point(10,20), Point(0,0))
>>> shapes = [p, c, s, r]
>>> for shape in shapes:
...     print(shape.area())
0 # Point p’s area
78.5 # Circle c’s area
36 # Square s’s area
```
Polymorphism refers to the fact that what an object does in response to a message (a method call) depends on the type or class of the object.

```python
class Shape:
    def area(self):
        return 'unknown'

class Point(Shape):
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def area(self):
        return 0

class Circle(Shape):
    def __init__(self, center, radius):
        self.center = center
        self.radius = radius
    def area(self):
        return (3.14 * self.radius * self.radius)

class Square(Shape):
    def __init__(self, length):
        self.length
    def area(self):
        return self.length * self.length

class Rectangle(Shape):
    def __init__(self, point1, point2):
        self.p1 = point1
        self.p2 = point2
    def area(self):
        length = self.p1.y - self.p2.y
        width = self.p2.x - self.p1.x
        return (length*width)
```

```python
>>> p = Point(10, 20)
>>> c = Circle(p, 5)
>>> s = Square(6)
>>> r = rectangle(Point(10,20), Point(0,0))
>>> shapes = [p, c, s, r]
>>> for shape in shapes:
...     print(shape.area())
0  # Point p’s area
78.5  # Circle c’s area
36  # Square s’s area
200  # Rectangle r’s area
```
QUESTIONS?