Week 5 - Lecture 1: OOP Part 2.
Today’s Menu
(Wrapping up OOP)

>> Inheritance
  - Employee and Student as subclasses of Person
  - `isinstance()` vs `type()`
  - `super()`
  - OOPy animation

>> Special methods of the form `__foo__( )`

>> Static methods, Class attributes
Inheritance: motivating example

You are the CMU president.

You have a program to keep track of people on campus.
class Employee(object):
    def __init__(self, name, age, gender, salary):
        self.name = name
        self.age = age
        self.gender = gender
        self.salary = salary
        self.rating = 3  # initial value out of 5

    def changeName(self, newName):
        # Some code to check if newName is valid.
        # if it is, then self.name = newName

    def changeAge(self, newAge):
        ...

    def changeRating(self, newRating):
        ...

    def changeSalary(self, newSalary):
        ...
def printInfo(self):
    print("Name:", self.name)
    print("Age:", self.age)
    print("Gender:", self.gender)
    print("Salary:", self.salary)
    print("Rating:", self.rating)

def getNetSalary(self):
    ...

def salaryInFuture(self, years):
    ...

# Add other methods
Inheritance: motivating example

```python
class Student(object):
    def __init__(self, name, age, gender, major):
        self.name = name
        self.age = age
        self.gender = gender
        self.major = major
        self.gpa = 0

    def changeName(self, newName):
        # Some code to check if newName is valid.
        # if it is, then self.name = newName

    def changeAge(self, newAge):
        ...

    def changeMajor(self, newMajor):
        ...

    def changeGpa(self, newGpa):
        ...
```

Inheritance: motivating example

```python
def printInfo(self):
    print("Name:", self.name)
    print("Age:", self.age)
    print("Gender:", self.gender)
    print("Major:", self.major)
    print("GPA:", self.gpa)

def isFailing(self):
    ...

# Add other methods
```
Inheritance: motivating example

Problems:

- code duplication
- not best way of structuring code
  > missing the overlap between Employee and Student
Employee and Student share:

Properties:

> name, age, gender

Methods:

> changeName, changeAge, printInfo

Why?

Employee and Student have a shared type. Both are a Person.
class Person(object):
    def __init__(self, name, age, gender):
        self.name = name
        self.age = age
        self.gender = gender

    def changeName(self, newName):
        # Some code to check if newName is valid.
        # if it is, then self.name = newName

    def changeAge(self, newAge):
        ...

    def printInfo(self):
        ...
Inheritance example

class Employee(Person):
    pass

Employee is now subclass of Person. Person is superclass of Employee.

Employee inherits every property and method of Person.

```python
e = Employee("Bob Marley", 26, "male")
e.printInfo()
```

```python
class Student(Person):
    pass

Student is now subclass of Person. Person is superclass of Student.

Student inherits every property and method of Person.

s = Student("Ada Lovelace", 21, "female")
s.changeAge(22)
```
Inheritance example

print(type(e))  <class '__main__.Employee'>
print(type(s))  <class '__main__.Student'>

print(type(e) == Employee)  True
print(type(s) == Student)  True

print(isinstance(e, Employee))  True
print(isinstance(s, Student))  True

print(isinstance(e, Person))  True
print(isinstance(s, Person))  True

print(type(e) == Person)  False
print(type(s) == Person)  False
class Employee(Person):
    def __init__(self, name, age, gender, salary):
        self.name = name
        self.age = age
        self.gender = gender
        self.salary = salary
        self.rating = 3

    def changeSalary(self, newSalary):
        ...

    def changeRating(self, newRating):
        ...

    def printInfo(self):
        ...

    def getNetSalary(self):
        ...

    def salaryInFuture(self, years):
        ...
class Employee(Person):
    def __init__(self, name, age, gender, salary):
        self.name = name
        self.age = age
        self.gender = gender
        self.salary = salary
        self.rating = 3

    def changeSalary(self, newSalary):
        ...

    def changeRating(self, newRating):
        ...

    def printInfo(self):
        ...

    def getNetSalary(self):
        ...

    def salaryInFuture(self, years):
        ...

changeName is inherited. (from Person class)
changeAge is inherited. (from Person class)
class Student(Person):
    def __init__(self, name, age, gender, major):
        self.name = name
        self.age = age
        self.gender = gender
        self.major = major
        self.gpa = 0

    def changeMajor(self, newMajor):
        ...

    def changeGpa(self, newGpa):
        ...

    def printInfo(self):
        ...

    def isFailing(self):
        ...

__init__ is overridden.
changeMajor is added.
changeGpa is added.
printInfo is overridden.
isFailing is added.
Inheritance: overriding methods

class Student(Person):
    def __init__(self, name, age, gender, major):
        self.name = name
        self.age = age
        self.gender = gender
        self.major = major
        self.gpa = 0
    def changeMajor(self, newMajor):
        ...
    def changeGpa(self, newGpa):
        ...
    def printInfo(self):
        ...
    def isFailing(self):
        ...

changeName is inherited.
(from Person class)

changeAge is inherited.
(from Person class)
Inheritance: avoiding code duplication

```python
class Person(object):
    def __init__(self, name, age, gender):
        self.name = name
        self.age = age
        self.gender = gender

...  

Code duplicated!

class Employee(Person):
    def __init__(self, name, age, gender, salary):
        self.name = name
        self.age = age
        self.gender = gender
        self.salary = salary
        self.rating = 3

...  
```
Inheritance: avoiding code duplication

```python
class Person(object):
    ...
    def printInfo(self):
        print("Name:", self.name)
        print("Age:", self.age)
        print("Gender:", self.gender)
    ...

class Employee(Person):
    ...
    def printInfo(self):
        print("Name:", self.name)
        print("Age:", self.age)
        print("Gender:", self.gender)
        print("Salary:", self.salary)
        print("Rating:", self.rating)
    ...
```

**Code duplicated!**

**NOTE:**
This is a simple example. In general, the duplicated code can be much longer and complex.
Inheritance: avoiding code duplication

```python
class Person(object):
    ...
    def printInfo(self):
        print("Name:" , self.name)
        print("Age:" , self.age)
        print("Gender:" , self.gender)
    ...

class Employee(Person):
    ...
    def printInfo(self):
        super().printInfo()
        print("Salary:" , self.salary)
        print("Rating:" , self.rating)
    ...
```
class Person(object):
    def __init__(self, name, age, gender):
        self.name = name
        self.age = age
        self.gender = gender

...

class Employee(Person):
    def __init__(self, name, age, gender, salary):
        super().__init__(name, age, gender)
        self.salary = salary
        self.rating = 3

...
Inheritance: another example

OOPy Animation
Today’s Menu
(Wrapping up OOP)

Inheritance

- **Employee** and **Student** as subclasses of **Person**
- `isinstance()` vs `type()`
- `super()`
- OOPy animation

>> Special methods of the form `__foo__( )`

>> Static methods, Class attributes
**object**: mother of all classes

```python
class Person(object):
    ...
```

**object** is actually a built-in data type (i.e. class).

When we define a class, we always make it a subclass of **object**.

What does **object** contain?

```python
>>> dir(object)
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__le__', '__lt__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__']
```
class Fraction(object):
    def __init__(self, num, den):
        self.num = num
        self.den = den
        self.simplify()

    def toString(self):
        return str(self.num) + "/" + str(self.den)

    def add(self, other):
        ...
    def mul(self, other):
        ...
    def toFloat(self):
        ...
    def simplify(self):
        ...

f1 = Fraction(4, 6)
f2 = Fraction(5, 9)
print(f1)  # <__main__.Fraction object at 0x1010349b0>
print(f1.toString())  # 2/3
print(f1.add(f2).toString())  # 11/9

print(f1.__str__())

print implicitly calls object’s __str__ method
Understanding methods \_\_\_foo\_\_( )

```python
class Fraction(object):
    def __init__(self, num, den):
        self.num = num
        self.den = den
        self.simplify()

    def __str__(self):
        return str(self.num) + "/" + str(self.den)

    def add(self, other):
        ...
    def mul(self, other):
        ...
    def toFloat(self):
        ...
    def simplify(self):
        ...

f1 = Fraction(4, 6)
f2 = Fraction(5, 9)
print(f1)  # 2/3
print(f1.add(f2))  # 11/9
print(f1.__str__())  # 2/3

print implicitly calls object's __str__ method
```
class Fraction(object):
    def __init__(self, num, den):
        self.num = num
        self.den = den
        self.simplify()

    def __str__(self):
        return str(self.num) + "/" + str(self.den)

    def __add__(self, other):
        ...
    def mul(self, other):
        ...
    def toFloat(self):
        ...
    def simplify(self):
        ...

f1 = Fraction(4, 6)
f2 = Fraction(5, 9)
print(f1)  # 2/3
print(f1 + f2)  # 11/9

+ implicitly calls object’s __add__ method
class Fraction(object):
    def __init__(self, num, den):
        self.num = num
        self.den = den
        self.simplify()

    def __str__(self):
        return str(self.num) + "/" + str(self.den)

    def __add__(self, other):
        ...
    def __mul__(self, other):
        ...
    def toFloat(self):
        ...
    def simplify(self):
        ...

f1 = Fraction(4, 6)
f2 = Fraction(5, 9)
print(f1)  # 2/3
print(f1 * f2)  # 10/27

* implicitly calls object’s __mul__ method
class Fraction(object):
    def __init__(self, num, den):
        self.num = num
        self.den = den
        self.simplify()

    def __str__(self):
        return str(self.num) + "/" + str(self.den)

    def __add__(self, other):
        ...
    def __mul__(self, other):
        ...
    def __float__(self):
        ...

f1 = Fraction(4, 6)
f2 = Fraction(5, 9)
print(f1)  # 2/3
print(float(f1))  # 0.6666666666666666

float implicitly calls object's __float__ method
Understanding methods `__foo__( )`

```
<   ___lt___
<=  ___le___
>   ___gt___
>=  ___ge___
==  ___eq___
!=  ___ne___
```
Be careful implementing these methods!

```python
def __eq__(self, other):
    return ((self.num == other.num) and (self.den == other.den))
```

```python
f1 = Fraction(4, 6)
f2 = Fraction(2, 3)
f3 = Fraction(2, 4)
print(f1 == f2)  # True
print(f1 == f3)  # False
print(f1 == 5)   # Crash
```

```python
def __eq__(self, other):
    return (isinstance(other, Fraction) and
            (self.num == other.num) and (self.den == other.den))
```
What if we try to put our objects in a set?

```python
f1 = Fraction(4, 6)
s = set()
s.add(f1)  # Either crashes, or doesn't work the way you want.
```

Built-in hash function calls the object’s `__hash__` method.

You need to override `__hash__` inherited from `object`:

```python
def __hash__(self):
    hashables = (self.num, self.den)
    return hash(hashables)
```
One annoying problem:

```python
f1 = Fraction(4, 6)
L = [f1]
print(L)  # [<__main__.Fraction object at 0x101e34a20>]
```

print actually calls `__repr__` for each element of the list.

So you should rewrite `__repr__`. 
## Summary

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__str__</code></td>
<td>Used by built-in <code>str</code> function</td>
</tr>
<tr>
<td><code>__repr__</code></td>
<td>To create computer readable form</td>
</tr>
<tr>
<td><code>__hash__</code></td>
<td>Used by built-in <code>hash</code> function</td>
</tr>
<tr>
<td><code>__float__</code></td>
<td>Used by built-in <code>float</code> function</td>
</tr>
<tr>
<td><code>__lt__</code></td>
<td>&lt;</td>
</tr>
<tr>
<td><code>__le__</code></td>
<td>&lt;=</td>
</tr>
<tr>
<td><code>__gt__</code></td>
<td>&gt;</td>
</tr>
<tr>
<td><code>__ge__</code></td>
<td>&gt;=</td>
</tr>
<tr>
<td><code>__eq__</code></td>
<td>==</td>
</tr>
</tbody>
</table>
Inheritance
- Employee and Student as subclasses of Person
- isinstance() vs type()
- super()
- OOPy animation

Special methods of the form __foo__( )

>> Static methods, Class attributes
class Fraction(object):
    def __init__(self, num, den):
        self.num = num
        self.den = den
        self.simplify()

    def simplify(self):
        g = gcd(self.num, self.den)
        self.num = self.num//g
        self.den = self.den//g

You might decide that you’ll only use gcd inside the Fraction class.

You might decide it belongs inside the Fraction class.

Yet, it can’t really be a method.

def gcd(a, b):
    while (b != 0):
        (a, b) = (b, a%b)
    return a
class Person(object):
    def __init__(self, name, age, gender):
        self.name = name
        self.age = age
        self.gender = gender

    def changeName(self, newName):
        if (isValidName(newName)):
            self.name = newName

    def changeAge(self, newAge):
        ...

    def isValidName(self, name):
        ...

isValidName is a helper function (and not a method).

We won’t really use it outside of Person class.

And we shouldn’t pollute the global space with it.
class Fraction(object):
    def __init__(self, num, den):
        self.num = num
        self.den = den
        self.simplify()

    def simplify(self):
        g = Fraction.gcd(self.num, self.den)
        self.num = self.num//g
        self.den = self.den//g

@staticmethod
def gcd(a, b):
    while (b != 0):
        (a, b) = (b, a%b)
    return a
class Person(object):
    def __init__(self, name, age, gender):
        self.name = name
        self.age = age
        self.gender = gender

    def changeName(self, newName):
        if (Person.isValidName(newName)):
            self.name = newName

    def changeAge(self, newAge):
        ...

@staticmethod
def isValidName(name):
    ...


Suppose we have a class called `Maze`.

```python
class Maze(object):
    def __init__(self):
        ...

    ...
```

Want to store directions:

```python
NORTH = (-1,0)
SOUTH = (1,0)
EAST  = (0,1)
WEST  = (0,-1)
```

- These are not really properties/fields of a maze.
- We are only going to use them in the `Maze` class.
- Every `Maze` object should share these variables.
class Maze(object):

    NORTH = (-1,0)
    SOUTH = (1,0)
    EAST  = (0,1)
    WEST  = (0,-1)

    def __init__(self):
        ...

    def solve(self, row,col):
        ...
        for drow,dcol in [Maze.NORTH, Maze.SOUTH, Maze.EAST, Maze.WEST]:
            ...

Note: NORTH, SOUTH, EAST, WEST are constants that don’t change.
Another example: back to dots demo.

class Dot(object):

    def __init__(self):
        ...

    ...

Want to store the total number of Dot instances created:

dotCount = 0

- This is not a property/field of a dot.

- Every Dot object should share this variable.

- Don’t want to pollute the global space.
class Dot(object):
    dotCount = 0

    def __init__(self):
        Dot.dotCount += 1

        ...

    ...

Properties/fields:
Every object-instance gets its own copy.

Class attributes:
There is only one copy (regardless of the number of instances).

Make it a class attribute.
Inheritance
- Employee and Student as subclasses of Person
- `isinstance()` vs `type()`
- `super()`
- OOPy animation

Special methods of the form `__foo__( )`

Static methods, Class attributes