Object Oriented Programming
Inheritance and polymorphism

Motivation

- Two types may have lots of similar code
- Reasons to avoid duplicated code:
  - More places to make, find and fix errors
  - Inconsistencies
  - Not lazy

Example: Employees

```python
class HourlyEmployee:
    def __init__(self, id, name, role, payRate, hours):
        self._id = id
        self._name = name
        self._role = role
        self._payRate = payRate
        self._hours = hours
    def __str__(self):
        # < code here >
    def computePay(self):
        # < code here >

class SalariedEmployee:
    def __init__(self, id, name, role, salary):
        self._id = id
        self._name = name
        self._role = role
        self._salary = salary
    def __str__(self):
        # < code here >
    def computePay(self):
        # < code here >
```

How might you make this better?

One generic Employee?

```python
class Employee:
    def __init__(self, id, name, role, type, salary, hours, payRate):
        self._id = id
        self._name = name
        self._role = role
        self.type = type
        if (self.type == 'salaried'):
            self.salary = salary
        else:
            # ...
```

Evaluation:
- Less code
- Only 1 class
- A salaried employee shouldn’t have hours
- Doesn’t scale

Inheritance

- Allows a new class to specialize an existing class
- Only specifies what is different
- New class is referred to as a subclass, derived class or child class
- Existing class is referred to as superclass, base class or parent class

Relationships

- “is-a” : instantiation
  - implemented using constructors
- “has-a” : aggregation
  - implemented using instance data
  - specifies container-contained relationship
- “is-a-kind-of” : inheritance
  - specializes the parent class
Example

- **Ship** - represents all space ships
- **CloakedShip** - is a kind of ship that cannot be detected with sensors

```
Ship

CloakedShip
```

“has-a” vs “is-a-kind-of”

- Beginning students sometimes want to use inheritance inappropriately
- Consider a **Ship** and a **PhotonTorpedo**
  - A PhotonTorpedo is not a Ship
  - A Ship is not a PhotonTorpedo
  - **Inheritance cannot be used!**
- A **Ship** “has-a” **PhotonTorpedo**, so the torpedo should be an instance variable of the **Ship** class

Which of the following correctly indicate a superclass-subclass pairing?

1. dog - cat
2. animal - dog
3. dog - animal
4. dog - tail
5. None of the above

Draw an appropriate inheritance structure for the following:

- Gala
- Orange
- Fruit
- Apple
- Fuji
- Carrot

What goes where?

- Common attributes and methods should be “pushed” up to the parent

```
Employee
  + myID
  + myName
  + myRole
  + toString()

HourlyEmployee
  + myPayrate
  + myHours
  + computePay()

SalariedEmployee
  + mySalary
  + computePay()
```

Mechanics of inheritance

- To inherit from another class, use the name of the parent class in parentheses after the declaration of the child:

  ```
  class Child(Parent):
       # ...
  ```

- When calling a method on an instance of the child class:
  - If the method has been defined in the child class, use that definition
  - Otherwise, look for the definition in the parent class
class A:
    def __init__(self, x):
        self.x = x
    def __str__(self):
        return str(self.x)

class B(A):
    def __init__(self, x):
        self.x = x * 2

b = B(5)
print(b)

What is the output of this code?
A. 5  
B. 10  
C. 510  
D. This will cause an error

The Power of Inheritance

• Allow parent class to be in charge of shared details
• Need to be able to refer to the parent!

class Parent:
    def __init__(self):
        self.x = 3

class Child(Parent):
    def __init__(self):
        super().__init__()  
        self.y = 10

Two approaches:

1. class Parent:
   def __init__(self):
       self.x = 3

class Child(Parent):
   def __init__(self):
       super().__init__()
       self.y = 10

2. class Parent:
   def __init__(self):
       self.x = 3

class Child(Parent):
   def __init__(self):
       self.x = 3
       super().__init__()
       self.y = 10

Textbook approach

Many online tutorials (and other textbooks)

Overriding Methods

• Defining a method in a child class using the same header/signature as in its parent class
• Allows the child to give a different implementation
• Common example: __str__ of child class wants to add information to the parent’s version

Example

class A:
    def __init__(self):
        print("A’s init")
        self.x = 10

class B(A):
    def __init__(self):
        print("B’s init")
        self.x = 10

b = B()
a = A()

print(b.x)

Person class

Person

Student

Professor

Tenured Professor
Polymorphism

- Function behaves differently depending on type of arguments
- An instance of a subclass can be passed as an argument to a function expecting an instance of a superclass

```python
class Wolf(object):
    def bark(self):
        print('Hooooowll)

class Dog(object):
    def bark(self):
        print('Woof')

def bark_for_me(dogtype):
    dogtype.bark()

my_dog = Dog()
my_wolf = Wolf()
bark_for_me(my_dog)
bark_for_me(my_wolf)
```

One last idea: Class Variables

- Used to track information shared by all members of a class
- Created inside the class, but not inside a method
- Can be accessed in two ways:
  - `classname.variable`
  - `objectname.variable`

Example

```python
class Person:
    def __init__(self, first, last):
        self.first = first
        self.last = last

class Student(Person):
    count = 0 # class variable!
    def __init__(self, first, last, year):
        super().__init__(first, last)
        self.year = year
        Student.count = Student.count + 1

#main code ...
print(Student.count)
a = Student('J', 'Doe', 2014)
print(a.count)
print(Student.count)
```