Inheritance in Python

Thomas Schwarz, SJ

Inheritance

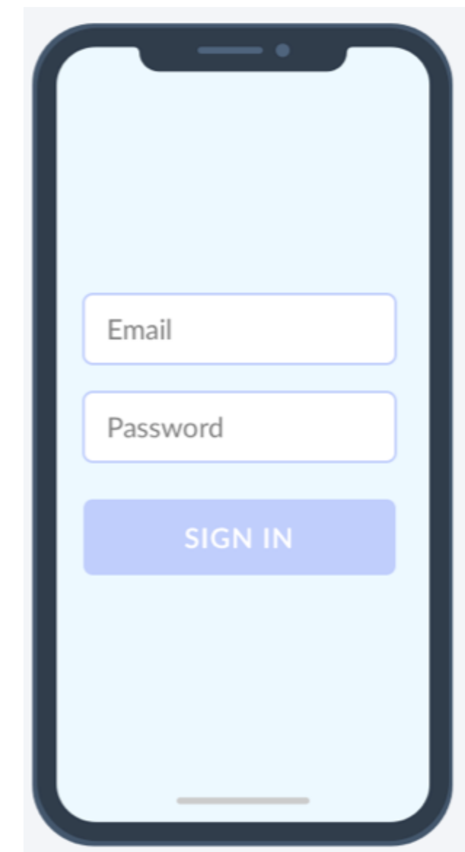
- Sometimes, classes have other classes as components:
  - Clients have addresses
    - Class Client has a field of type Class Address
- Sometimes, classes expand other classes
  - Example: animal -> dog -> poodle
    - The poodle is a dog, the dog is an animal
  - Example:
    - employee -> engineer (an engineer is an employee)
    - employee -> first level manager (a manager is an employee)

Inheritance

- The manager and the employee share data and functionality
  - If we implement them as classes:
    - Manager Class and Engineer Class have common fields and common methods.
- This is a common phenomenon

Inheritance

- Graphics implementation:
  - An app has a number of elements
    - Buttons, Canvases, Labels, EntryBoxes, Icons, ...
  - All these elements share:
    - The idea of size (usually a rectangle in the app)
    - Certain functionality



Inheritance

- We have identified two possible relationships between classes
  - **is_a**
    - objects of one class are also instances of another class
      - Poodles are Dogs
  - **has_a**
    - objects of one class are fields (aka properties aka members) of another class

Inheritance

- These are of course not the only relationships between classes
    - Methods can have arguments that are objects of different classes
    - Methods can use one class as an argument and return an instance of another class
    - etc

Inheritance

- We implement the common structure in a
  - *Base Class* (aka. parent class)
- We implement the specifics in a
  - *Derived Class* (aka child class)

Inheritance

- Example:
- Class Poodle is derived from Class Dog
- Class Dog is derived from Class Animal

Inheritance

- How do we do it:
  - We first implement the parent class
  - We then implement the child class
  - We derive by putting the name of the parent in parenthesis in the definition of the child class

```
class Parent:
    ...

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

Inheritance

- Example: Base Class is Person.
    - A person has a name and a birthdate
- Derive a class Employee
    - An employee is a person
    - An employee has an annual salary

Inheritance

- Implement the base class (minimum):

```
class Person:
    def __init__(self, name, birthday):
        self.name = name
        self.birthday = birthday
    def __str__(self):
        return '{} (born {})'.format(self.name,
      self.birthday)

if __name__ == '__main__':
    abe = Person('Abraham Lincoln', 'Feb 12, 1809')
    doug = Person('Stephen Douglas', 'Apr 23, 1813')
    bell = Person('John Bell', 'Feb 18, 1796')
    print(abe, doug, bell)
```

Inheritance

- To derive the child class:
  - In the constructor, add a call to the parent class constructor
  - Then add new fields / properties

```
class Employee(Person):
    def __init__(self, name, birthday, salary):
        Person.__init__(self, name, birthday)
        self.salary = salary
```

Inheritance

- Instead of calling the constructor of the parent class by name, we can also use the `super` method
  - super( ) automatically gets the Parent class
    - There is no self - parameter in the call

```
class Employee(Person):
    def __init__(self, name, birthday, salary):
        super().__init__(name, birthday)
        self.salary = salary
```

Method Overriding

- In our implementation, we now have
  - two __init__ dunder methods
  - two __str__ dunder methods
- This is called **method overriding**
- Any object has a type, in this case, a class
  - Depending on the object's class, the right method is invoked

Method Overriding

- Self-test:
  - Create a dunder hash for Person, composed of the hash for name and birthday
  - Create a dunder hash for Employee, composed of the hashes of person and the birthday

Selftest Solution

```
class Person:
...
def __hash__(self):
        return hash(self.name)+hash(self.birthday)
```

```
class Employee(Person):
    ...
    def __hash__(self):
        return hash(self.name)+hash(self.birthday)
+self.salary
```

Private Members of a Parent Class

- Many programming languages allow to make fields (aka properties) private
    - The "private parts" joke
- Python does not use a compiler to enforce privacy
- In line with Perl:
    - “Perl doesn't have an infatuation with enforced privacy. It would prefer that you stayed out of its living room because you weren't invited, not because it has a shotgun” — Larry Wall

Private Members of a Parent Class

- Python enforces rules by convention
  - Convention 1: If you want other programmers or yourself to leave the fields in a class alone, you preface them with a single underscore
  - Convention 2: If you want to be 'embarrassingly private', use double underscores before

Private Members of a Parent Class

- Python enforces the double underscore rule by mangling
  - Internally, properties with an initial double underscore are stored under a different name
  - But the name is predictable, so you **can** break the rule after all
  - But it would be very impolite
    - Either making them private was a bad idea
    - Or breaking privacy is horribly bad

Private Members of a Parent Class

- Let's change Person to have a private property
  - I cannot thing of anything that makes sense, so lets use a nonsensical property code

```
class Person:
    def __init__(self, name, birthday):
        self.name = name
        self.birthday = birthday
        self.__code = 'P'
```

Private Members of a Parent Class

- If I try to access it directly, I get an error:

```
>>> abe = Person('Abraham Lincoln', 'Feb 12, 1809')
>>> abe.__code
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    abe.__code
AttributeError: 'Person' object has no attribute
'__code'
```

Private Members of a Parent Class

- But I can access it by using the mangled name:
  - Mangler calls the field _<class name>__field

```
>>> abe._Person__code
'P'
```

Comparison with other languages

- Object Oriented programming was introduced with two big advantages in mind:
  - Code Reusability
    - You do not need to re-implement a class from another project
  - Modularity
    - Simpler design
    - Containment of errors: Easier to pinpoint a class implementation at fault
- These promises have been only partially fulfilled.

Comparison with other languages

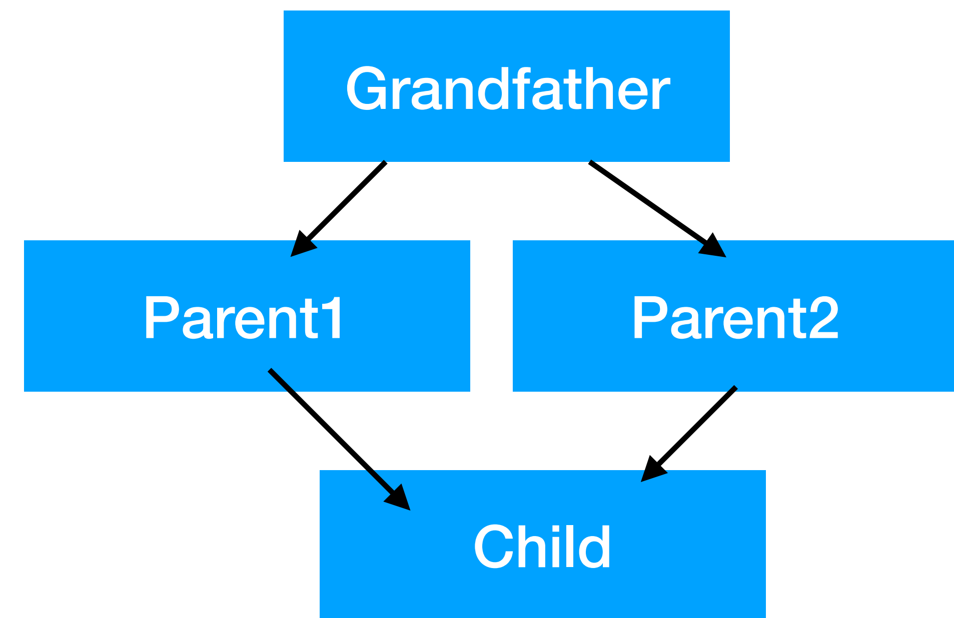
- Code reuse:
  - Rarely happens in practice other than through the implementation of libraries.
  - For easier code reuse, C++ uses templates
    - E.g. one list instead of list of integers, list of strings, etc.
    - Python does this through 'duck typing'
      - As long as something behaves like a duck, it is a duck

Comparison with other languages

- Modularity
  - C++, Java enforce access restrictions
    - These can be circumvented with dirty tricks
    - Force programmers to redeclare fields as private, protected, public
  - Python uses protection by "convention", not protection by compiler error
    - If you want to, but you should not want to, you can declare fields private using the double underscore

Comparison with other languages

- Code Reuse:
  - Inheritance allows us to reuse code written for a base class
  - Inheritance becomes difficult when the diamond pattern is allowed:
  - What happens if parents share a method with the same name
  - What if one parent overwrites a grandfather method and the other one does not



Comparison with other languages

- Multiple Inheritance: a class derives from more than one class
  - Not allowed in Java, but allowed in Python and C++
    - If used, need to understand how Python resolves names of methods and fields
-

Comparison with other languages

- Interfaces:
  - a type of class interface used in Java to assure that classes fulfill certain requirements
    - e.g. a class implementing an interface has a hash method
- Python can use "Abstract Base Classes" to provide the same support
  - Advanced topic

Comparison with other languages

- Python OO is easy if you stick with the basics
- If you want to do advanced stuff, there is more to learn
-