

Inheritance

Oliver W. Layton

CS151: Computational Thinking: Visual Media

Lecture 30, Spring 2021

Monday April 26

Inheritance: Parent and child classes

Technique to **share code** and **reduce code redundancy** when designing a class highly related to one you already have.

- **Parent class:** Class that you have that is a "broad thing".
- **Child class:** Class that you are designing that is a "more specific thing".

Child class inherits methods from parent class

Child class and objects you make out of it get a **copy** of all the **methods** in the parent (as if you copy-pasted the parent class methods into the child class).

Called **inheriting** parent's methods.

- Example: `lecture30_veggie_inherits.py`
- Method actually stored in the parent class, but the child class and its object have a connection ("arrow") to it.
- Methods defined in child class, but not in parent class, only useable in child class (not parent).
- Let's draw the symbol table for `lecture30_veggie_inherits.py`.

Is-a relationship

Inheritance works well when an object of the child class **is a** thing that the parent class represents.

- Example: A CS Student (child) **is a** Colby Student (parent)
- Example: A gas car (child) **is a** vehicle (parent)
- Example: Earth (child) **is a** planet (parent)
- The more specific thing is usually the child, the more general thing is usually the parent.

Inheritance terminology

Parent class

Child class

Base class

Derived class

Superclass

Subclass

Pairs of terms tend to go together, but mean the same thing.

Overriding a method

The child class can **override** any of the methods of the parent (create a method with the same name as method in parent, but have it do different things).

- The overridden method gets called when called on a child object.
- The parent method gets called when called on the parent object.
- Example: define in *both* parent and child.
 - `describeYourself(self)` called on Veggie prints savory.
 - `describeYourself(self)` called on SweetPotato prints sweet.
- Example: `lecture_31_veggie_override.py` then let's draw the symbol table.

Extending a method

If method exists in both child and parent, child can call the parent's version (e.g. to do redundant operations). Example: Constructor:

```
class SweetPotato(Veggie):  
    def __init__(self, name):  
        Veggie.__init__(self, name)
```

- Can also call the parent method from the child method using:
(super().<method>(param1,param2,...))
- Think of code in parent method copy-pasted where it is called from child method.
- Calling parent method and possibly doing additional work / assigning instance variables unique to child is called **extending** the method.
- Example: lecture_31_veggie_extend.py

Overriding vs. Extending (1/2)

When we **override** a parent method, the child **replaces** its behavior:

```
class Veggie:
    def can_eat_raw(self):
        return True
```

```
class SweetPotato(Veggie):
    def can_eat_raw(self):
        return False
```

Overriding vs. Extending (2/2)

When we **extend** a parent method, we make use of parent method and possibly do work unique to child:

```
class Veggie:
    def __init__(self, name):
        self.name = name

class SweetPotato(Veggie):
    def __init__(self, name, isAYam):
        super().__init__(name)
        self.isAYam = isAYam
```

Instance variables are NOT automatically inherited

To have the same set of instance variables as parent, we need to call the parent's constructor in child constructor.

- Pass along shared/overlapping parameters to parent constructor.
- Example: `lecture31_veggie_ivs.py`

Another example: CSStudent class

Let's design a CSStudent student class to inherit from a ColbyStudent class.

Benefits of Inheritance

- **Promotes code reuse.** Methods defined in parent (Veggie) also implemented for child class (SweetPotato) (e.g. getName()).
- Saves time programming / fewer lines of code.
- Less duplicate code **makes your code less error prone** / easier to debug
 - The more times you write essentially the same method, the more bugs you could introduce.
- Promotes **encapsulation** best practices.
 - Only the designer of the parent and child class needs to know how each is implemented.
 - From the outside, users that create objects of the parent and child types do not need to know that they are related by inheritance. They both can be used just like regular classes.