Discussion 7
Inheritance and Interfaces

CS61A Sections 29/47

Dickson Tsai

dickson.tsai+cs61a@berkeley.edu

dicksontsai.com/meet

OH: Tu/Th 4-5pm 411 Soda

Anuncios

* Project 3 due next Thursday 10/23
* CS61A Hackathon this weekend. Show your skills, or practice your skills!

* Midterm 2 next next Monday 10/27
  * Prepare for it now! Play around with the interpreter/ write your own programs/master the basics!
  * Remember: all programs (simple and complex) follow the same rules

Prev Stop: Mutable Data/Nonlocal

* Can mutate lists/dictionaries by accessing them
* Can mutate nonlocal frame values by using the `nonlocal` keyword

* But we want to create our own objects!
* `nonlocal` only modifies closest frame with the name

Current Stop: Inheritance/Interfaces

* Solution: Syntax for defining own types (classes) and creating objects of those types -> OOP

* Objects have attributes (key-value pairs) much like frames
* Focus on defining **similarities and differences** via class inheritance and instance/class attribute distinction
* Interfaces = type-independent abstraction. Many different classes can match attributes.

Next Stop: Trees (v2)/Efficiency

* Trees: Practice OOP with different types of trees
* Efficiency: Can affect how you want to represent your data/implement your program

Section Quiz

* Printed out for you. Please turn them in once you are done.

Your Feedback from Last Week

* Spend more time talking about problems in the back, not just the front
  * Will do
* Suggest problems for good midterm practice
  * Check out cs61a.org/problems
* Real life applications
* Go over process and thinking + group work
* Exam level problems
  * Should only be attempted when easy content is mastered. Try creating your own questions/solutions.

Section Quiz [Solutions]

Draw all the objects that will be created.
Write out their attributes.

```
class A:
  def __init__(self, num):
    candy = num * 2
    self.num = num
a = A(3)
b = A(4)
```

class	A
__init__	-> func __init__(self, num)

instance	A
num	3

instance	A
num	4

- The class will be an object as well! __init__ is an attribute of that obj.
- candy is not an attribute. It is a local variable that never gets added to the object. self.candy would add the candy attribute to the instances.

Section Quiz [Solutions]

What would Python print?

class	A
depths	[1, None]
__init__	-> func __init__(self, num)

```
>>> class A:
...     depths = [1, print(2)]
...     def __init__(self, num):
...       self.num = num
2
>>> A.__init__
function
>>> a = A(3)
(Nothing gets printed)
```

instance	A
num	3

```
>>> A.num = 2
>>> a.num
3
>>> A.depths
[1, None]
>>> a.__init__
Bound method
>>> a.depths
[1, None]
```

Section Quiz [Solutions]

Write a class that will pass this doctest: *Hint: What data should you keep track of & where?*

```
>>> a = A(3)
>>> a.run(2)
[3, 3]
>>> a.run(5)
[3, 3, 3, 3, 3]
>>> A.mul = 2
>>> a.run(2)
[3, 3, 3, 3]
```

Data to keep track of:

- The number in the list
- The number of times run
- The multiplier

Where should they go?

Consider: instance attribute, class attribute, etc.

Number in list -> instance attribute
Multiplier -> class attribute
Number of times run -> function parameter

Section Quiz [Solutions]

Write a class that will pass this doctest: *Hint: What data should you keep track of & where?*

```
>>> a = A(3)
>>> a.run(2)
[3, 3]
>>> a.run(5)
[3, 3, 3, 3, 3]
>>> A.mul = 2
>>> a.run(2)
[3, 3, 3, 3]
```

Number in list -> instance attribute
Multiplier -> class attribute
Number of times run -> function parameter

```
class A:
  mul = 1
  def __init__(self, num):
    self.num = num
  def run(self, times):
    return [self.num for i in range(times \
                * self.mul)]
```

OOP: Initial Attempt

Inheritance assertion:
Nothing in common.

```
class Dog(object):
  def __init__(self, name, owner, color):
    self.name = name
    self.owner = owner
    self.color = color
  def eat(self, thing):
    print(self.name + " ate " + thing)
  def talk(self):
    return self.name + " says woof!"
```

```
class Cat(object):
  def __init__(self, name, owner, lives):
    self.name = name
    self.owner = owner
    self.lives = lives
  def eat(self, thing):
    print(self.name + " ate " + thing)
  def talk(self):
    return self.name + " says meow!"
```

OOP: Inheritance Motivation

* Inheritance so that you just specify **similarities and differences** among *hierarchical* object types
* Dog and Cat objects are also Pet instances

```
class Pet(object):
  sound = “”
  def __init__(self, name, owner):
    self.name = name
    self.owner = owner
def eat(self, thing):
    print(self.name + " ate " + thing)
  def talk(self):
    return self.name + " says ” + self.sound
```

- All pets (including dogs and cats) have owners
- All pets eat
- All pets talk

This example differs slightly from the discussion example

OOP: Inheritance Motivation

* Then, just specify **differences** in the subclass.

```
class Dog(Pet):
  sound = 'woof!'
  def __init__(self, name, owner, color):
    Pet.__init__(self, name, owner)
    self.color = color
```

- Dogs also need to store color
- Dog's eat is same as Pet's eat, so don't include at all
- Dog's talk like Pet's talk. Just change the sound!

```
class Cat(Pet):
  sound = 'meow!'
  def __init__(self, name, owner, lives):
    Pet.__init__(self, name, owner)
    self.lives = lives
```

- Cats also need to store lives
- Cat's eat is same as Pet's eat, so don't include at all
- Cat's talk like Pet's talk. All cats "meow!", so add class attr

OOP: High Level View

* So, classes describe an object (data+behavior) generally.
  * That way, to construct a specific object, fill in right parameters
  * Then, when the class changes, all objects of that class change with it
* Where have we seen this?
  * Functions describe a computation generally
  * Use parameters to carry out specific computation
* And inheritance captures similarities/differences between classes, making classes easier to describe.

OOP Reminders

Object Creation

1. Start with empty instance
2. Initialize the empty instance (add initial data to it, e.g. data it must have to even be considered a Person, such as gender)
3. The object can now be used

* __init__ = function meant to **fill in** an empty object of class. Does not create the object.

Functions vs. Bound methods

1. Method = function that belongs to a class object (not instance)
2. Bound method = created every time a method is looked up from an instance, e.g. fido.eat
   * Different from function in that self is already bound
   * E.g. func eat(self, thing) → bound method fido.eat(thing)

The self keyword

* Idea: Each method should be able to access the instance of the class easily
  * Thus, assume that self will be instance of the class.

Example

```
class A:
  def __init__(self, num):
    self.num = num
```

self is assumed to be an instance of A

Any needed info that's not in the instance/class should be a parameter. Otherwise, just access that info from **self**!

Tackling OOP Implementation Problems

Representation

1. Really put thought into what data you need to remember and where you should remember it:
   * In the class? Each instance? As method local variables?
   * Does it make conceptual sense?
     * Each human has different height. It doesn't make sense to put height as a class attribute.

Communication

1. Remember that objects are meant to communicate with each other
   a. Which methods are outside objects supposed to invoke?
   b. Did you separate responsibilities correctly?
      * E.g. Chess GUI manager should not know chess rules like en passant. That's for the Chess Referee object.

OOP Code Tips

1. When using inheritance, remember to keep in mind **similarities and differences**.
   * If your code is too similar to another part of your program, and you are using inheritance, you probably want to re-evaluate the similarities.
     * Chances are, you can refactor your code.
   * Remember that you can inherit class attributes. As much as possible, data should not be mixed with computation in inheritance
     * Any hard coding you do can probably be saved as attributes.

Interfaces

Interface – shared set of attributes with specification of attribute's behavior

Example: **__str__** interface

* Any objects should present their data readably
  * __str__(self) – human-readable representation
* Any object that has these attributes can now have non-default output printed using the **print()** function.

Interfaces

Q: How are interfaces different from inheritance?

- Different objects can support a certain set of data/behavior but otherwise are unrelated
- Can't specify everything in terms of similarities/differences as in inheritance.

Example: datetimes and trees have __str__ attribute. Both can be printed specially. Apart from __str__ interface, not related.

Interfaces vs. Inheritance re-explained

Advantages:

- Yet another form of data abstraction. Don't care how the value of the attributes are implemented
- Only select a certain part to be shared, other parts kept the same.
  - Inheritance means everything must be shared or overridden.

Disadvantages:

* In Python, no particular syntax for interfaces, so must be documented separately