CS 61A Summer 2025

OOP, Inheritance, String Representation

Discussion 6: July 17, 2025

Object-Oriented Programming

Before starting the discussion, you can read over the introduction/refresher for OOP below!

Object-oriented programming (OOP) uses objects and classes to organize programs. Here's an example of a class:

```
class Car:
    max_tires = 4

    def __init__(self, color):
        self.tires = Car.max_tires
        self.color = color

    def drive(self):
        if self.tires < Car.max_tires:
            return self.color + ' car cannot drive!'
        return self.color + ' car goes vroom!'

    def pop_tire(self):
        if self.tires > 0:
            self.tires -= 1
```

Class: The type of an object. The `Car` class (shown above) describes the characteristics of all `Car` objects.

Object: A single instance of a class. In Python, a new object is created by calling a class.

```
>>> ferrari = Car('red')
```

Here, `ferrari` is a name bound to a `Car` object.

Class attribute: A variable that belongs to a class and is accessed via dot notation. The `Car` class has a `max_tires` attribute.

```
>>> Car.max_tires
4
```

Instance attribute: A variable that belongs to a particular object. Each `Car` object has a `tires` attribute and a `color` attribute. Like class attributes, instance attributes are accessed via dot notation.

```
>>> ferrari.color
'red'
>>> ferrari.tires
4
>>> ferrari.color = 'green'
>>> ferrari.color
'green'
```

Method: A function that belongs to an object and is called via dot notation. By convention, the first parameter of a method is `self`.

When one of an object's methods is called, the object is implicitly provided as the argument for `self`. For example, the `drive` method of the `ferrari` object is called with empty parentheses because `self` is implicitly bound to the `ferrari` object.

```
>>> ferrari = Car('red')
>>> ferrari.drive()
'red car goes vroom!'
```

We can also call the original `Car.drive` function. The original function does not belong to any particular `Car` object, so we must provide an explicit argument for `self`.

```
>>> ferrari = Car('red')
>>> Car.drive(ferrari)
'red car goes vroom!'
```

init: A special function that is called automatically when a new instance of a class is created.

Notice how the `drive` method takes in `self` as an argument, but it looks like we didn't pass one in! This is because the dot notation *implicitly* passes in `ferrari` as `self` for us. So in this example, `self` is bound to the object called `ferrari` in the global frame.

To evaluate the expression `Car('red')`, Python creates a new `Car` object. Then, Python calls the `__init__` function of the `Car` class with `self` bound to the new object and `color` bound to `'red'`.

Inheritance lets us define relationships between classes. Consider the following `Dog` and `Cat` classes:

```
class Dog:
    def __init__(self, name, owner):
        self.is_alive = True
        self.name = name
        self.owner = owner
    def eat(self, thing):
        print(self.name + " ate a " + str(thing) + "!")
    def talk(self):
        print(self.name + " says woof!")

class Cat:
    def __init__(self, name, owner, lives=9):
        self.is_alive = True
        self.name = name
        self.owner = owner
        self.lives = lives
    def eat(self, thing):
        print(self.name + " ate a " + str(thing) + "!")
    def talk(self):
        print(self.name + " says meow!")
```

Dogs and cats have a lot in common, so there is a lot of repeated code! To avoid redefining shared attributes and methods, we can write a single *base class* that is *inherited* by more specific classes.

For example, we can write a `Pet` class that serves as the base class for `Dog`:

```
class Pet:
    def __init__(self, name, owner):
        self.is_alive = True
        self.name = name
        self.owner = owner
    def eat(self, thing):
        print(self.name + " ate a " + str(thing) + "!")
    def talk(self):
        print(self.name)

class Dog(Pet):
    def talk(self):
        print(self.name + ' says woof!')
```

Inheritance lets us indicate that one class is a more specific version of another. Each dog is a pet; in other words, `Dog` is a *subclass* of `Pet`, and `Pet` is a *superclass* of `Dog`.

We don't have to redefine `__init__` or `eat` because `Dog` inherits those functions from `Pet`. However, we want each dog to talk in a way that is unique to dogs, so we have the `Dog` class *override* the `talk` function.

We can also rewrite the `Cat` class as a subclass of `Pet`:

```
class Cat(Pet):
    def __init__(self, name, owner, lives=9):
        super().__init__(name, owner)
        self.lives = lives
    def talk(self):
        print(self.name + " says meow!")
```

The `super()` expression lets us access the functions of a superclass (e.g. `Pet`) within the functions of a subclass (e.g. `Cat`).

In the `__init__` function of the `Cat` class, `super()` evaluates to a version of `self` that behaves as if it were a `Pet` instead of a `Cat`.

The expression `super().__init__(name, owner)` is equivalent to `Pet.__init__(self, name, owner)`. By calling `super().__init__`, we avoid rewriting the code that assigns `is_alive`, `name`, and `owner` attributes.

Hint: A good way to get started with defining a class is to determine the necessary class attributes and instance attributes. Before implementing a class, describe the type of each attribute and how it will be used.

Q1: Keyboard

Overview: A keyboard has a button for every letter of the alphabet. When a button is pressed, it outputs its letter by calling an `output` function (such as `print`). Whether that letter is uppercase or lowercase depends on how many times the *caps lock* key has been pressed.

First, implement the `Button` class, which takes a lowercase `letter` (a string) and a one-argument `output` function, such as `Button('c', print)`.

The `press` method of a `Button` calls its `output` attribute (a function) on its `letter` attribute: either uppercase if `caps_lock` has been pressed an odd number of times or lowercase otherwise. The `press` method also increments `pressed` and returns the key that was pressed. *Hint*: `'hi'.upper()` evaluates to `'HI'`.

Second, implement the `Keyboard` class. A `Keyboard` has a dictionary called `keys` containing a `Button` (with its `letter` as its key) for each letter in `LOWERCASE_LETTERS`. It also has a list of the letters `typed`, which may be a mix of uppercase and lowercase letters.

The `type` method takes a string `word` containing only lowercase letters. It invokes the `press` method of the `Button` in `keys` for each letter in `word`, which adds a letter (either lowercase or uppercase depending on `caps_lock`) to the `Keyboard`'s `typed` list. **Important:** Do not use `upper` or `letter` in your implementation of `type`; just call `press` instead.

Read the doctests and talk about:

- Why it's possible to press a button repeatedly with `.press().press().press()`.
- Why pressing a button repeatedly sometimes prints on only one line and sometimes prints multiple lines.
- Why `bored.typed` has 10 elements at the end.

Note: This worksheet is a problem bank—most TAs will not cover all the problems in discussion section.

```
LOWERCASE_LETTERS = 'abcdefghijklmnopqrstuvwxyz'

class CapsLock:
    def __init__(self):
        self.pressed = 0

    def press(self):
        self.pressed += 1

class Button:
    """A button on a keyboard.

    >>> f = lambda c: print(c, end='')  # The end='' argument avoids going to a new line
    >>> k, e, y = Button('k', f), Button('e', f), Button('y', f)
    >>> s = e.press().press().press()
    eee
    >>> caps = Button.caps_lock
    >>> t = [x.press() for x in [k, e, y, caps, e, e, k, caps, e, y, e, caps, y, e, e]]
    keyEEKeyeYEE
    >>> u = Button('a', print).press().press().press()
    A
    A
    A
    """
    caps_lock = CapsLock()

    def __init__(self, letter, output):
        assert letter in LOWERCASE_LETTERS
        self.letter = letter
        self.output = output
        self.pressed = 0

    def press(self):
        """Call output on letter (maybe uppercased), then return the button that was
    pressed."""
        self.pressed += 1
        "*** YOUR CODE HERE ***"
```

```
class Keyboard:
    """A keyboard.

    >>> Button.caps_lock.pressed = 0  # Reset the caps_lock key
    >>> bored = Keyboard()
    >>> bored.type('hello')
    >>> bored.typed
    ['h', 'e', 'l', 'l', 'o']
    >>> bored.keys['l'].pressed
    2

    >>> Button.caps_lock.press()
    >>> bored.type('hello')
    >>> bored.typed
    ['h', 'e', 'l', 'l', 'o', 'H', 'E', 'L', 'L', 'O']
    >>> bored.keys['l'].pressed
    4
    """
    def __init__(self):
        self.typed = []
        self.keys = ...  # Try a dictionary comprehension!

    def type(self, word):
        """Press the button for each letter in word."""
        assert all([w in LOWERCASE_LETTERS for w in word]), 'word must be all lowercase'
        "*** YOUR CODE HERE ***"
```

Q2: Shapes

Fill out the skeleton below for a set of classes used to describe geometric shapes. Each class has an `area` and a `perimeter` method, but the implementation of those methods is slightly different. Please override the base `Shape` class's methods where necessary so that we can accurately calculate the perimeters and areas of our shapes with ease.

```python
class Shape:
    """All geometric shapes will inherit from this Shape class."""
    def __init__(self, name):
        self.name = name

    def area(self):
        """Returns the area of a shape"""
        print("Override this method in ", type(self))

    def perimeter(self):
        """Returns the perimeter of a shape"""
        print("Override this function in ", type(self))

class Circle(Shape):
    """A circle is characterized by its radii"""
    def __init__(self, name, radius):
        "*** YOUR CODE HERE ***"




    def perimeter(self):
        """Returns the perimeter of a circle (2r)"""
        "*** YOUR CODE HERE ***"



    def area(self):
        """Returns the area of a circle (r^2)"""
        "*** YOUR CODE HERE ***"
```

```
class RegPolygon(Shape):
    """A regular polygon is defined as a shape whose angles and side lengths are all the
    same.
    This means the perimeter is easy to calculate. The area can also be done, but it's
    more inconvenient."""
    def __init__(self, name, num_sides, side_length):
        "*** YOUR CODE HERE ***"






    def perimeter(self):
        """Returns the perimeter of a regular polygon (the number of sides multiplied by
    side length)"""
        "*** YOUR CODE HERE ***"


class Square(RegPolygon):
    def __init__(self, name, side_length):
        "*** YOUR CODE HERE ***"



    def area(self):
        """Returns the area of a square (squared side length)"""
        "*** YOUR CODE HERE ***"



class Triangle(RegPolygon):
    """An equilateral triangle"""
    def __init__(self, name, side_length):
        "*** YOUR CODE HERE ***"



    def area(self):
        """Returns the area of an equilateral triangle is (squared side length multiplied
     by the provided constant"""
        constant = math.sqrt(3)/4
        "*** YOUR CODE HERE ***"
```

Q3: Bear

Implement the `SleepyBear` and `WinkingBear` classes so that calling their `print` method matches the doctests. Use as little code as possible and try not to repeat any logic from `Eye` or `Bear`. Each blank can be filled with just two short lines.

```
class Eye:
    """An eye.

    >>> Eye().draw()
    '0'
    >>> print(Eye(False).draw(), Eye(True).draw())
    0 -
    """
    def __init__(self, closed=False):
        self.closed = closed

    def draw(self):
        if self.closed:
            return '-'
        else:
            return '0'

class Bear:
    """A bear.

    >>> Bear().print()
    ? 0o0?
    """
    def __init__(self):
        self.nose_and_mouth = 'o'

    def next_eye(self):
        return Eye()

    def print(self):
        left, right = self.next_eye(), self.next_eye()
        print('? ' + left.draw() + self.nose_and_mouth + right.draw() + '?')
```

```python
class SleepyBear(Bear):
    """A bear with closed eyes.

    >>> SleepyBear().print()
    ? -o-?
    """
    "*** YOUR CODE HERE ***"




class WinkingBear(Bear):
    """A bear whose left eye is different from its right eye.

    >>> WinkingBear().print()
    ? -oO?
    """
    def __init__(self):
        "*** YOUR CODE HERE ***"




    def next_eye(self):
        "*** YOUR CODE HERE ***"
```

You're done! Excellent work this week. Please be sure to fill out your TA's attendance form to get credit for this discussion!