CS 61A Summer 2026

OOP, Inheritance, String Representation

Discussion 6: July 16, 2026

Object-Oriented Programming

Object-oriented programming (OOP) is a programming paradigm that allows us to treat data as objects, like we do in real life.

For example, consider the **class** `Student`. Each of you as individuals is an **instance** of this class.

Details that all CS 61A students have, such as `name`, are called **instance variables**. Every student has these variables, but their values differ from student to student. A variable that is shared among all instances of `Student` is known as a **class variable**. For example, the `extension_days` attribute is a class variable as it is a property of all students.

All students are able to do homework, attend lecture, and go to office hours. When functions belong to a specific object, they are called **methods**. In this case, these actions would be methods of `Student` objects.

Here is a recap of what we discussed above:

- **class**: a template for creating objects
- **instance**: a single object created from a class
- **instance variable**: a data attribute of an object, specific to an instance
- **class variable**: a data attribute of an object, shared by all instances of a class
- **method**: a bound function that may be called on all instances of a class

Instance variables, class variables, and methods are all considered **attributes** of an object.

To avoid redefining attributes and methods for similar classes, we can write a single **base class** from which more specialized classes **inherit**. For example, we can write a class called `Pet` and define `Dog` as a **subclass** of `Pet`:

```
class Pet:
    def __init__(self, name, owner):
        self.is_alive = True    # It's alive!!!
        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):
        super().talk()
        print('This Dog says woof!')
```

Inheritance represents a hierarchical relationship between two or more classes where one class **is a** more specific version of the other: a dog **is a** pet. (We use "**is a**" to describe this sort of relationship in OOP languages, not to refer to the Python `is` operator.)

Since `Dog` inherits from `Pet`, the `Dog` class will also inherit the `Pet` class's methods, so we don't have to redefine `__init__` or `eat`. We do want each `Dog` to `talk` in a `Dog`-specific way, so we can **override** the `talk` method.

We can use `super()` to refer to the superclass of `self`, and access any superclass methods as if we were an instance of the superclass. For example, `super().talk()` in the `Dog` class will call the `talk` method from the `Pet` class, but passes in the `Dog` instance as the `self`.

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. Do not use `append` in your implementation of `type` either; you must mutate the list `typed` only using `press`.

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.

Since `self.letter` is always lowercase, use `self.letter.upper()` to produce the uppercase version.

The number of times `caps_lock` has been pressed is either `self.caps_lock.pressed` or `Button.caps_lock.pressed`.

The `output` attribute is a function that can be called: `self.output(self.letter)` or `self.output(self.letter.upper())`. You do not need to return the result.

```
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 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
        if self.caps_lock.pressed % 2 == 1:
            self.output(self.letter.upper())
        else:
            self.output(self.letter)
        return self
```

The keys can be created using dictionary comprehension: `self.keys = {c: Button(c, ...) for c in LETTERS}`. The call to `Button` should take `c` and **an output function that appends to `self.typed`**, so that every time one of these buttons is pressed, it appends a letter to `self.typed`.

Call the `press` method of `self.key[w]` for each `w` in `word`. It should be the case that when you call `press`, the `Button` is already set up (in the `Keyboard.__init__` method) to output to the `typed` list of this `Keyboard`.

```
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 = []
        # END SOLUTION
        # BEGIN SOLUTION NO PROMPT ALT="# Try a dictionary comprehension!"
        self.keys = {c: Button(c, self.typed.append) for c in LOWERCASE_LETTERS}

    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'
        for w in word:
            self.keys[w].press()
```

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.
Reminder: You can import the **math** class for math.pi.

```
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):
        super().__init__(name)
        self.radius = radius



    def perimeter(self):
        """Returns the perimeter of a circle  (2r)"""
        return 2*pi*self.radius



    def area(self):
        """Returns the area of a circle  (r^2)"""
        return pi*self.radius**2
```

```
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):
        super().__init__(name)
        self.num_sides = num_sides
        self.side_length = side_length




    def perimeter(self):
        """Returns the perimeter of a regular polygon (the number of sides multiplied by
    side length)"""
        return self.num_sides*self.side_length


class Square(RegPolygon):
    def __init__(self, name, side_length):
        super().__init__(name, 4, side_length)



    def area(self):
        """Returns the area of a square (squared side length)"""
        return self.side_length**2



class Triangle(RegPolygon):
    """An equilateral triangle"""
    def __init__(self, name, side_length):
        super().__init__(name, 3, side_length)



    def area(self):
        """Returns the area of an equilateral triangle is (squared side length multiplied
     by the provided constant"""
        constant = math.sqrt(3)/4
        return constant*self.side_length**2
```

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

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

class Bear:
    """A bear.

    >>> Bear().print()
    ? OoO?
    """
    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() + '?')
```

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

    >>> SleepyBear().print()
    ? -o-?
    """
    def next_eye(self):
        return Eye(True)




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

    >>> WinkingBear().print()
    ? -o0?
    """
    def __init__(self):
        super().__init__()
        self.eye_calls = 0




    def next_eye(self):
        self.eye_calls += 1
        return Eye(self.eye_calls % 2)
```

String Representations

We've learned how to use classes to represent data as objects, but sometimes we need to a string representation of the object. We can customize how objects are displayed by defining the special methods **str** and **repr** in our class.

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

peng = Penguin(3, 'Peng')
```

Let's use this Penguin class as an example! Right now, if we try to output or print **peng** out, we get this:

```
>>> peng
<__main__.Penguin object at 0x104cb9c40>
>>> print(peng)
<__main__.Penguin object at 0x104cb9c40>
```

That doesn't tell me a lot about peng! We use **str** and **repr** to output a more understandable representation.

str is the "informal" representation used as an output to represent an object. This is outputed when we call print() or str() on our object, like str(peng) or print(peng).

```
def __str__(self):
    return self.name + ' is a ' + str(self.age) + ' year old penguin!'
>>> str(peng)
'Peng is a 3 year old penguin!'
>>> print(peng)
Peng is a 3 year old penguin!
```

repr returns the formal string representation of an object, often useful for deubgging, and shows up when displaying the object directly, like putting peng in the interpreter.

```
def __repr__(self):
    return "Penguin(Name=\'" + self.name + "\', age=" + str(self.age)+")"
>>> peng
"Penguin(Name = Peng, Age = 3)"
```

Q4: WWPD: repr and str

Important: For all WWPD questions, type `Function` if you believe the answer is `<function...>`, `Error` if it errors, and `Nothing` if nothing is displayed.

```
>>> print("hi")
```

hi

```
>>> "hi"
```

'hi'

```
>>> print(repr("hi"))
```

'hi'

```
>>> repr("hi")
```

" 'hi' "

```
>>> class A:
...     def __init__(self, x):
...         self.x = x
...     def __repr__(self):
...         return self.x
>>> class B(A):
...     def __str__(self):
...         return self.x + self.x
>>> A("hi")
```

hi

```
>>> print(A("hi"))
```

hi

```
>>> B("hi")
```

hi

```
>>> print(B("hi"))
```

hihi

```
>>> class C:
...     def __str__(self):
...         print('hi')
...         return 'hihi'
...     def __repr__(self):
...         print('hihihi')
...         return 'hihihihi'
>>> C()
```

hihihi hihihihi

```
>>> print(C())
```

hi hihi

```
>>> q = str(C())
```

hi

```
>>> q
```

'hihi'

```
>>> r = repr(C())
```

hihihi

```
>>> r
```

'hihihihi'