DATA C88C
Fall 2025

Object-Oriented Programming, Inheritance

Discussion 7: October 20, 2025

Object-Oriented Programming

A productive approach to defining new classes is to determine what instance attributes each object should have and what class attributes each class should have. First, describe the type of each attribute and how it will be used, then try to implement the class's methods in terms of those attributes.

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.

Discussion Time: Before anyone types anything, have a conversation describing the type of each attribute and how it will be used. Start with `Button`: how will `letter` and `output` be used? Then discuss `Keyboard`: how will `typed` and `keys` be used? How will new letters be added to the list called `typed` each time a `Button` in `keys` is pressed? Call the staff if you're not sure! Once everyone understands the answers to these questions, you can try writing the code together.

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

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.

```
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 = {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()
```

The keys can be created using a 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`.

Description Time: Describe how new letters are added to `typed` each time a `Button` in `keys` is pressed. Instead of just reading your code, say what it does (e.g., "When the button of a keyboard is pressed …"). One short sentence is enough to describe how new letters are added to `typed`.

Inheritance

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`.

Q2: Cat

Below is the implementation of a `Pet` class. Each pet has two instance attributes (`name` and `owner`), as well as one instance method (`talk`).

```
class Pet:

    def __init__(self, name, owner):
        self.name = name
        self.owner = owner

    def talk(self):
        print(self.name)
```

Implement the `Cat` class, which inherits from the `Pet` class seen above. To complete the implementation, override or implement the following methods:

`___init___`

Set the `Cat`'s `name` and `owner` attributes, and also add 2 new attributes:

1. `is_hungry` - should be set to `False`
2. `fullness` - should be set to whatever the `fullness` parameter is

   Hint: You can call the `__init__` method of `Pet` (the superclass of `Cat`) to set a cat's `name` and `owner` using `super()`.

`talk`

Print out a cat's greeting, which is `"<name of cat> says meow!"`.

`get_hungry`

Decrements a cat's `fullness` level by 1. When `fullness` reaches zero, `is_hungry` becomes `True`. If this is called after `fullness` has already reached zero, print the message `"<name of cat> is hungry."`

`eat`

This method is called when the cat eats some food.

If the cat is hungry, after calling this method both of the following should be true:

1. The cat's `fullness` value should be set to whatever `Cat.default_fullness` is.
2. The cat's `is_hungry` value should be `False`.

Also print out the food the cat ate. For example, if a cat named Thomas ate fish, print out `'Thomas ate a fish!'`

Otherwise, if the cat wasn't hungry, print `'<name of cat> is not hungry.'`

```
class Cat(Pet):
    default_fullness = 5

    def __init__(self, name, owner, fullness=default_fullness):
        """
        >>> cat = Cat('Thomas', 'Tammy')
        >>> cat.name
        'Thomas'
        >>> cat.owner
        'Tammy'
        >>> cat.fullness # use default fullness value
        5
        >>> cat.is_hungry
        False
        >>> cat2 = Cat('Meow Meow', 'Yoobin', 3)
        >>> cat2.fullness # use fullness value that was passed in
        3
        """
        super().__init__(name, owner)
        self.fullness = fullness
        self.is_hungry = False

    def talk(self):
        """
        >>> Cat('Thomas', 'Tammy').talk()
        Thomas says meow!
        >>> Cat('Meow Meow', 'ThuyAnh').talk()
        Meow Meow says meow!
        """
        print(self.name + ' says meow!')

    def get_hungry(self):
        """
        >>> cat = Cat('Thomas', 'Tammy', 2)
        >>> cat.is_hungry
        False
        >>> cat.fullness
        2
        >>> cat.get_hungry()
        >>> cat.is_hungry
        False
        >>> cat.fullness
        1
        >>> cat.get_hungry()
        >>> cat.is_hungry
        True
        >>> cat.fullness
        0
        >>> cat.get_hungry()
        Thomas is hungry.
        >>> cat.is_hungry
        True
```