CS 61A Spring 2026

OOP, Inheritance

Discussion 7: March 11, 2026

Attendance

Your TA will come around during discussion to check you in. You can start on the worksheet before being checked in; you don't need to wait for your TA to get started.

Getting Started

Say your name and something you've practiced for a while, such as playing an instrument, juggling, or martial arts. Did you discover any common interests among your group members?

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.

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 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`.

```
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()
```

Discussion 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`.

Q2: 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()
    ? -oO?
    """
    def __init__(self):
        super().__init__()
        self.eye_calls = 0




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

Object-oriented programming problems often appear on exams.

Q3: Counter

Fall 2024 Final Exam Question 4(a): Implement the `Counter` class. A `Counter` has a `count` of the number of times `inc` has been invoked on itself or any of its offspring. Its offspring are the Counters created by its `spawn` method or the `spawn` method of any of its offspring.

```
class Counter:
    """Counts how many times inc has been called on itself or any of its spawn.

    >>> total = Counter()
    >>> odd, even = total.spawn(), total.spawn()
    >>> one, three = odd.spawn(), odd.spawn()
    >>> for c in [one, even, three, even, odd, even]:
    ...     c.inc()
    >>> [c.count for c in [one, three, even, odd, total]]
    [1, 1, 3, 3, 6]
    """
    def __init__(self, parent=None):
        self.parent = parent
        self.count = 0

    def inc(self):
        self.count += 1
        if self.parent is not None:
            self.parent.inc()

    def spawn(self):
        return Counter(self)
```

Q4: MissDict

Fall 2024 Final Exam Question 4(b): Implement the `MissDict` class. A `MissDict` has a dictionary `d`. Its `get` method takes an iterable `keys`, returns a list of all values in `d` that correspond to those `keys`, and counts the number of `keys` that did not appear in `d` (called *misses*). Printing a `MissDict` displays a fraction in which:

- The numerator is the number of misses during all calls to `get` for that particular `MissDict` instance.
- The denominator is the number of misses during all calls to `get` for any `MissDict` instance.

Assume `Counter` is implemented correctly.

```
class MissDict:
    """Has a dict, gets a list of values for an iterable of keys, and counts keys that
    are not in the dict.

    >>> double = MissDict({1: 2, 2: 4, 3: 6, 5: 10})
    >>> half = MissDict({2: 1.0, 3: 1.5, 4: 2.0})
    >>> double.get([1, 3, 5, 2, 4])  # No value for key 4 (1 miss)
    [2, 6, 10, 4]
    >>> double.get([5, 4, 3, 0, 4])  # No value for keys 0 or either 4 (3 misses)
    [10, 6]
    >>> half.get([1, 3, 5, 2, 4])    # No value for keys 1 or 5 (2 misses)
    [1.5, 1.0, 2.0]
    >>> print(double)
    4/6 of the misses
    """
    misses = Counter()
    def __init__(self, d):
        assert isinstance(d, dict)
        self.d = d
        self.misses = MissDict.misses.spawn()

    def get(self, keys):
        result = []
        for k in keys:
            if k in self.d:
                result.append(self.d[k])
            else:
                self.misses.inc()
        return result

    def __str__(self):
        return f'{self.misses.count}/{MissDict.misses.count} of the misses'
```

Optional Questions

For more practice with iterators!

Q5: Draw

The `draw` function takes a list `hand` and a list of unique non-negative integers `positions` that are all less than the length of `hand`. It removes `hand[p]` for each `p` in `positions` and returns a list of those elements in the order they appeared in `hand` (not the order they appeared in `positions`).

Fill in each blank with one of these names: `list`, `map`, `filter`, `reverse`, `reversed`, `sort`, `sorted`, `append`, `insert`, `index`, `remove`, `pop`, `zip`, or `sum`. See the built-in functions and list methods documentation for descriptions of what these do.

For a list `s` and integer `i`, `s.pop(i)` returns and removes the `i`th element, which changes the position (index) of all the later elements but does not affect the position of prior elements.

Calling `reversed(s)` on a list `s` returns an iterator. Calling `list(reversed(s))` returns a list of the elements in `s` in reversed order.

```
def draw(hand, positions):
    """Remove and return the items at positions from hand.

    >>> hand = ['A', 'K', 'Q', 'J', 10, 9]
    >>> draw(hand, [2, 1, 4])
    ['K', 'Q', 10]
    >>> hand
    ['A', 'J', 9]
    """
    return list(reversed( [hand.pop(i) for i in reversed(sorted(positions))] ))
```

Aced it? Give yourselves a *hand*!