CS 61A Fall 2023 — OOP

Discussion 7: March 6, 2024

Pick someone in your group to join Discord. It's fine if multiple people join, but one is enough.

Now switch to Pensieve:

- **Everyone**: Go to discuss.pensieve.co and log in with your @berkeley.edu email, then enter your group number. (Your group number is the number of your Discord channel.)

Once you're on Pensieve, you don't need to return to this page; Pensieve has all the same content (but more features). If for some reason Penseive doesn't work, return to this page and continue with the discussion.

Post in the `#help` channel on Discord if you have trouble.

Getting Started

Say your name, another class you're taking besides CS 61A, 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?

Q1: 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.

Discussion Time: Before writing anything, talk as a group about what process you'll implement in order to make sure the right cards are removed and returned. Try not to guess-and-check! The purpose of discussion is for you to try to solve problems without the help of an interpreter checking your work.

```
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 _____(_____( [hand._____(i) for i in _____(_____(positions))] ))
```

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.

Aced it? Give yourselves a *hand*!

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.

Q2: 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
        "*** YOUR CODE HERE ***"









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



```
    >>> Button.caps_lock.pressed = 0  # Reset the caps_lock key
    >>> bored = Keyboard()
```

Please don't look at the hints until you've discussed as a group and agreed that you need a hint.

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. Instead return `self` afterward in order to return the button that was pressed.

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`.

Presentation 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`. When you're ready, send a message to the `#discuss-queue` channel with the `@discuss` tag, your discussion group number, and the message "Put it on our tab!" and a member of the course staff will join your voice channel to hear your description and give feedback.

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.

Discussion Time: Before writing code, talk about what is different about a `SleepyBear` and a `Bear`. When using inheritance, you only need to implement the differences between the base class and subclass. Then, talk about what is different about a `WinkingBear` and a `Bear`. Can you think of a way to make the bear wink without a new implementation of `print`?

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

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

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

class Bear:
    """A bear.

    >>> Bear().print()
      • •
    """
    def __init__(self):
        self.nose_and_mouth = ' '

    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()
       --
    """
    "*** YOUR CODE HERE ***"




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

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

Implement a `next_eye` method that returns an Eye instance that is closed.

One way to make the bear wink is to track how may times the `next_eye` method is invoked using a new instance attribute and return a closed eye if `next_eye` has been called an even number of times.

Document the Occasion

Please all fill out the attendance form (one submission per person per week).