CS 61A
Fall 2023

OOP, String Representation

Discussion 7: October 11, 2023

Getting Started

Someone from your group should join Discord.

To get help from a TA, send a message to the `discuss-queue` channel with the @discuss tag and your discussion group number.

If you have only 1 or 2 people in your group, you can join the other group in the room with you.

The most common suggestion from last discussion was to make discussion more fun. Therefore, we've added some puns to the presentation prompts. Have fun!

Q1: Ice Breaker

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.

Q2: 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`.

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.

```
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*. You're a big *deal*. Then write the return expression of `draw` in your group's Discord channel's text chat.

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.

Q3: Keyboard

Implement the `Button` class, which takes a lowercase `letter` and a one-argument `output` function. When `pressed`, a `Button` calls its `output` function on a version of its `letter`: 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.

Then, implement the `Keyboard` class. A `Keyboard` has a list of the letters `typed` and a dictionary called `keys` containing a `Button` value (with its `letter` as its key) for each letter in `LOWERCASE_LETTERS`.

The `type` method takes a string `word` containing only letters. It invokes the `press` method of the `Button` in `keys` for each letter in `s`, which adds a letter (either lowercase or uppercase depending on `caps_lock`) to the `Keyboard`'s `typed` list. It returns a list of the elements of `typed` that were added as a result. Do not use `upper` or `letter` in `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 `typed` each time a `Button` in `keys` is pressed? Once everyone understands the answers to these questions, you can try writing the code together.

Recommended: If you want feedback on your understanding and approach before you start writing code, message "preparation is key" and your group number to `@discuss` in the `#discuss-queue` channel.

```
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 a version of letter, 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')
```

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(...) 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`.

Store the length of `self.typed` before pressing any keys, then press `self.key[w]` for each `w` in `word`, then return a slice of `self.typed` that only includes the letters just added (everything after its original length).

Presentation Time: *Return* to describing how new letters are added to `typed` each time a `Button` in `keys` is pressed. Not all descriptions are *equal*: one *option* is to just read your code out loud, but *escape* that temptation and *shift* toward describing your program in terms of objects. (E.g., "When the button of a keyboard is pressed …") One short sentence is enough to describe how new letters are added to `typed`, but you're in *control*; use more *space* if you want. 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.

Q4: Bear

Implement the `Eye`, `Bear`, `SleepyBear`, and `WinkingBear` classes so that printing these objects matches the doctests. Use as little code as possible and try not to repeat any logic. Each blank can be filled with just one or 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 `__str__`?

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

    >>> Eye().draw_eye()
    •''
    >>> print(Eye(), Eye(True), Eye())
    • - •
    >>> str(Eye()) != repr(Eye())
    True
    """
    def __init__(self, closed=False):
        self.closed = closed

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

    "*** YOUR CODE HERE ***"




class Bear:
    """A bear.

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

    def eye(self):
        return Eye()

    def __str__(self):
        return '  ' + str(self.eye()) + self.nose_and_mouth + str(self.eye()) + ' '

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

    >>> print(SleepyBear())
       --
    """
    "*** YOUR CODE HERE ***"
```

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

```
class WinkingBear(Bear):
```

Implement a `__str__` method, which is called automatically when an object is printed.

Implement an `eye` method that returns an Eye instance.

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

Presentation Time: Once your group thinks you've solved the problem with as little code as possible, paste your whole implementation into your group's Discord channel's text chat, then send a message to the `#discuss-queue` channel with the `@discuss` tag, your discussion group number, and the message " • • " and a member of the course staff will join your voice channel to check over your code and give you suggestions.

Document the Occasion

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

If anyone scored points in the slap-the-table-when-someone-says-class game, post the highest score in your group's Discord channel's text chat.