CS 61A Sequences, Mutability, Object-Oriented Programming

Spring 2022 Discussion 5: February 23, 2022 Solutions

##Walkthrough Videos

Feel free to try these problems on the worksheet in discussion or on your own, and then come back to reference these walkthrough videos as you study.

> To see these videos, you should be logged into your berkeley.edu email.

Sequences

Sequences are ordered collections of values that support element-selection and have length. We've worked with lists, but other Python types are also sequences, including strings.

Q1: Map, Filter, Reduce

Many languages provide `map`, `filter`, `reduce` functions for sequences. Python also provides these functions (and we'll formally introduce them later on in the course), but to help you better understand how they work, you'll be implementing these functions in the following problems.

> In Python, the `map` and `filter` built-ins have slightly different behavior than the `my_map` and `my_filter` functions we are defining here.

`my_map` takes in a one argument function `fn` and a sequence `seq` and returns a list containing `fn` applied to each element in `seq`.

```python
def my_map(fn, seq):
    """Applies fn onto each element in seq and returns a list.
    >>> my_map(lambda x: x*x, [1, 2, 3])
    [1, 4, 9]
    """
    result = []
    for elem in seq:
        result += [fn(elem)]
    return result
```

`my_filter` takes in a predicate function `pred` and a sequence `seq` and returns a list containing all elements in `seq` for which `pred` returns True.

```
def my_filter(pred, seq):
    """Keeps elements in seq only if they satisfy pred.
    >>> my_filter(lambda x: x % 2 == 0, [1, 2, 3, 4])  # new list
    has only even-valued elements
    [2, 4]
    """
    result = []
    for elem in seq:
        if pred(elem):
            result += [elem]
    return result
```

`my_reduce` takes in a two argument function `combiner` and a non-empty sequence `seq` and combines the elements in `seq` into one value using `combiner`.

```
def my_reduce(combiner, seq):
    """Combines elements in seq using combiner.
    seq will have at least one element.
    >>> my_reduce(lambda x, y: x + y, [1, 2, 3, 4])  # 1 + 2 + 3 + 4
    10
    >>> my_reduce(lambda x, y: x * y, [1, 2, 3, 4])  # 1 * 2 * 3 * 4
    24
    >>> my_reduce(lambda x, y: x * y, [4])
    4
    >>> my_reduce(lambda x, y: x + 2 * y, [1, 2, 3]) # (1 + 2 * 2) +
     2 * 3
    11
    """
    total = seq[0]
    for elem in seq[1:]:
        total = combiner(total, elem)
    return total
```

Mutability

Some objects in Python, such as lists and dictionaries, are **mutable**, meaning that their contents or state can be changed. Other objects, such as numeric types, tuples, and strings, are **immutable**, meaning they cannot be changed once they are created.

Let's imagine you order a mushroom and cheese pizza from La Val's, and they represent your order as a list:

```
>>> pizza = ['cheese', 'mushrooms']
```

With list mutation, they can update your order by mutate `pizza` directly rather than having to create a new list:

```
>>> pizza.append('onions')
>>> pizza
['cheese', 'mushrooms', 'onions']
```

Aside from `append`, there are various other list mutation methods:

- `append(el)`: Add `el` to the end of the list. Return `None`.
- `extend(lst)`: Extend the list by concatenating it with `lst`. Return `None`.
- `insert(i, el)`: Insert `el` at index `i`. This does not replace any existing elements, but only adds the new element `el`. Return `None`.
- `remove(el)`: Remove the first occurrence of `el` in list. Errors if `el` is not in the list. Return `None` otherwise.
- `pop(i)`: Remove and return the element at index `i`.

We can also use list indexing with an assignment statement to change an existing element in a list. For example:

```
>>> pizza[1] = 'tomatoes'
>>> pizza
['cheese', 'tomatoes', 'onions']
```

Q2: WWPD: Mutability

What would Python display? In addition to giving the output, draw the box and pointer diagrams for each list to the right.

```
>>> s1 = [1, 2, 3]
>>> s2 = s1
>>> s1 is s2
```

True

```
>>> s2.extend([5, 6])
>>> s1[4]
```

6

```
>>> s1.append([-1, 0, 1])
>>> s2[5]
```

[-1, 0, 1]

```
>>> s3 = s2[:]
>>> s3.insert(3, s2.pop(3))
>>> len(s1)
```

5

```
>>> s1[4] is s3[6]
```

True

```
>>> s3[s2[4][1]]
```

1

```
>>> s1[:3] is s2[:3]
```

False

```
>>> s1[:3] == s2[:3]
```

True

```
>>> s1[4].append(2)
>>> s3[6][3]
```

2

OOP

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 `max_slip_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.

Q3: WWPD: Student OOP

Below we have defined the classes `Professor` and `Student`, implementing some of what was described above. Remember that Python passes the `self` argument implicitly to methods when calling the method directly on an object.

```
class Student:

    max_slip_days = 3 # this is a class variable

    def __init__(self, name, staff):
        self.name = name # this is an instance variable
        self.understanding = 0
        staff.add_student(self)
        print("Added", self.name)

    def visit_office_hours(self, staff):
        staff.assist(self)
        print("Thanks, " + staff.name)

class Professor:

    def __init__(self, name):
        self.name = name
        self.students = {}

    def add_student(self, student):
        self.students[student.name] = student

    def assist(self, student):
        student.understanding += 1

    def grant_more_slip_days(self, student, days):
        student.max_slip_days = days
```

What will the following lines output?

```
>>> callahan = Professor("Callahan")
>>> elle = Student("Elle", callahan)
```

Added Elle

```
>>> elle.visit_office_hours(callahan)
```

Thanks, Callahan

```
>>> elle.visit_office_hours(Professor("Paulette"))
```

Thanks, Paulette

```
>>> elle.understanding
```

2

```
>>> [name for name in callahan.students]
```

['Elle']

```
>>> x = Student("Vivian", Professor("Stromwell")).name
```

Added Vivian

```
>>> x
```

'Vivian'

```
>>> [name for name in callahan.students]
```

['Elle']

```
>>> elle.max_slip_days
```

3

```
>>> callahan.grant_more_slip_days(elle, 7)
>>> elle.max_slip_days
```

7

```
>>> Student.max_slip_days
```

3

Q4: Keyboard

We'd like to create a `Keyboard` class that takes in an arbitrary number of `Buttons` and stores these `Buttons` in a dictionary. The keys in the dictionary will be ints that represent the postition on the `Keyboard`, and the values will be the respective

`Button`. Fill out the methods in the `Keyboard` class according to each description, using the doctests as a reference for the behavior of a `Keyboard`.

```
class Button:
    def __init__(self, pos, key):
        self.pos = pos
        self.key = key
        self.times_pressed = 0

class Keyboard:
    """A Keyboard takes in an arbitrary amount of buttons, and has a
    dictionary of positions as keys, and values as Buttons.
    >>> b1 = Button(0, "H")
    >>> b2 = Button(1, "I")
    >>> k = Keyboard(b1, b2)
    >>> k.buttons[0].key
    'H'
    >>> k.press(1)
    'I'
    >>> k.press(2) # No button at this position
    ''
    >>> k.typing([0, 1])
    'HI'
    >>> k.typing([1, 0])
    'IH'
    >>> b1.times_pressed
    2
    >>> b2.times_pressed
    3
    """
    def __init__(self, *args):
        self.buttons = {}
        for button in args:
            self.buttons[button.pos] = button

    def press(self, info):
        """Takes in a position of the button pressed, and
        returns that button's output."""
        if info in self.buttons.keys():
            b = self.buttons[info]
            b.times_pressed += 1
            return b.key
        return ''

    def typing(self, typing_input):
        """Takes in a list of positions of buttons pressed, and
        returns the total output."""
        accumulate = ''
        for pos in typing_input:
            accumulate+=self.press(pos)
        return accumulate
```