CS 61A Summer 2022

Inheritance, Midterm Review

Discussion 6: July 12, 2022

Midsemester Survey

Please fill out the course-wide midsemester survey by Wednesday, 7/13 @ 11:59pm PT. This survey is designed to help us make short term adjustments to the course so that it works better for you. We appreciate your feedback. We may not be able to make every change that you request, but we will read all the feedback and consider it.

Inheritance

Python classes can implement a useful abstraction technique known as *inheritance*. To illustrate this concept, consider the following `Dog` and `Cat` classes.

```
class Dog():
    def __init__(self, name, owner):
        self.is_alive = True
        self.name = name
        self.owner = owner
    def eat(self, thing):
        print(self.name + " ate a " + str(thing) + "!")
    def talk(self):
        print(self.name + " says woof!")

class Cat():
    def __init__(self, name, owner, lives=9):
        self.is_alive = True
        self.name = name
        self.owner = owner
        self.lives = lives
    def eat(self, thing):
        print(self.name + " ate a " + str(thing) + "!")
    def talk(self):
        print(self.name + " says meow!")
```

Notice that because dogs and cats share a lot of similar qualities, there is a lot of repeated code! To avoid redefining attributes and methods for similar classes, we can write a single *base class* from which the similar classes *inherit*. For example, we can write a class called `Pet` and redefine `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):
        print(self.name + ' says woof!')
```

Inheritance represents a hierarchical relationship between two or more classes where one class *is a* (no relation to the Python `is` operator) more specific version of the other, e.g. a dog *is a* pet. Because `Dog` inherits from `Pet`, we didn't have to redefine `__init__` or `eat`. However, since we want `Dog` to `talk` in a way that is unique to dogs, we did *override* the `talk` method.

We can use the `super()` function to refer to a class's *superclass*. For example, calling `super()` within the class definition of `Dog` allows us to access the same object but as if it were an instance of its superclass, in this case `Pet`. This is a little bit of a simplification, and if you're interested you can read more here.

Here's an example of an alternate equivalent definition of `Dog` that uses `super()` to explicitly call the `__init__` method of the parent class:

```
class Dog(Pet):
    def __init__(self, name, owner):
        super().__init__(name, owner)
        # this is equivalent to calling Pet.__init__(self, name,
    owner)
    def talk(self):
        print(self.name + ' says woof!')
```

Keep in mind that creating the `__init__` function shown above is actually not necessary, because creating a `Dog` instance will automatically call the `__init__` method of `Pet`. Normally when defining an `__init__` method in a subclass, we take some additional action to calling `super().__init__`. For example, we could add a new instance variable like the following:

```
def __init__(self, name, owner, has_floppy_ears):
    super().__init__(name, owner)
    self.has_floppy_ears = has_floppy_ears
```

Q1: Cat

Below is a skeleton for the `Cat` class, which inherits from the `Pet` class. To complete the implementation, override the `__init__` and `talk` methods and add a new `lose_life` method.

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

```python
class Cat(Pet):

    def __init__(self, name, owner, lives=9):
        "*** YOUR CODE HERE ***"










    def talk(self):
        """Print out a cat's greeting.

        >>> Cat('Thomas', 'Tammy').talk()
        Thomas says meow!
        """
        "*** YOUR CODE HERE ***"






    def lose_life(self):
        """Decrements a cat's life by 1. When lives reaches zero,
        is_alive becomes False. If this is called after lives has
        reached zero, print 'This cat has no more lives to lose.'
        """
        "*** YOUR CODE HERE ***"













# You can use more space on the back if you want
```

Q2: NoisyCat

More cats! Fill in this implementation of a class called `NoisyCat`, which is just like a normal `Cat`. However, `NoisyCat` talks a lot: in fact, it talks twice as much as a regular `Cat`! If you'd like to test your code, feel free to copy over your solution to the `Cat` class above.

```
class __________ # Fill me in!
    """A Cat that repeats things twice."""
    def __init__(self, name, owner, lives=9):
        # Is this method necessary? Why or why not?
        "*** YOUR CODE HERE ***"







    def talk(self):
        """Talks twice as much as a regular cat.
        >>> NoisyCat('Magic', 'James').talk()
        Magic says meow!
        Magic says meow!
        """
        "*** YOUR CODE HERE ***"






# You can use more space on the back if you want
```

Higher Order Functions

Please refer back to Discussion 2 for an overview of higher order functions.

Q3: High Score

For the purposes of this problem, a *score function* is a pure function which takes a single number `s` as input and outputs another number, referred to as the *score* of `s`. Complete the `best_k_segmenter` function, which takes in a positive integer `k` and a score function `score`.

`best_k_segmenter` returns a function that takes in a single number `n` as input and returns the *best k-segment* of `n`, where a *k-segment* is a set of consecutive digits obtained by segmenting `n` into pieces of size `k` and the best segment is the segment with the highest score as determined by `score`. The segmentation is **right to left**.

For example, consider `1234567`. Its 2-segments are `1`, `23`, `45` and `67` (a segment may be shorter than `k` if `k` does not divide the length of the number; in this case, `1` is the leftover, since the segmenation is right to left). Given the score function `lambda x: -x`, the best 2-segment is `1`. With `lambda x: x`, the best segment is `67`.

```python
def best_k_segmenter(k, score):
    """
    >>> largest_digit_getter = best_k_segmenter(1, lambda x: x)
    >>> largest_digit_getter(12345)
    5
    >>> largest_digit_getter(245351)
    5
    >>> largest_pair_getter = best_k_segmenter(2, lambda x: x)
    >>> largest_pair_getter(12345)
    45
    >>> largest_pair_getter(245351)
    53
    >>> best_k_segmenter(1, lambda x: -x)(12345)
    1
    >>> best_k_segmenter(3, lambda x: (x // 10) % 10)(192837465)
    192
    """
    partitioner = lambda x: (________________, ________________)
    def best_getter(n):
        assert n > 0
        best_seg = None
        while __________________:
            n, seg = partitioner(n)
            if __________________:
                best_seg = seg
        return __________________
    return __________________
```

Recursion

Please refer back to Discussion 3 for an overview of recursion.

Q4: Ten-pairs

Write a function that takes a positive integer `n` and returns the number of ten-pairs it contains. A ten-pair is a pair of digits within `n` that sums to 10. **Do not use any assignment statements.**

The number 7,823,952 has 3 ten-pairs. The first and fourth digits sum to 7+3=10, the second and third digits sum to 8+2=10, and the second and last digit sum to 8+2=10. Note that a digit can be part of more than one ten-pair.

> *Hint*: Complete and use the helper function `count_digit` to calculate how many times a digit appears in `n`.

```
def ten_pairs(n):
    """Return the number of ten-pairs within positive integer n.
    >>> ten_pairs(7823952)
    3
    >>> ten_pairs(55055)
    6
    >>> ten_pairs(9641469)
    6
    """
    "*** YOUR CODE HERE ***"











def count_digit(n, digit):
    """Return how many times digit appears in n.
    >>> count_digit(55055, 5)
    4
    """
    "*** YOUR CODE HERE ***"
```

Tree Recursion

Please refer back to Discussion 3 for an overview of tree recursion.

Q5: Making Onions

Write a function `make_onion` that takes in two one-argument functions, `f` and `g`, and applies them in layers (like an onion). `make_onion` is a higher-order function that returns a function that takes in three parameters, `x`, `y`, and `limit`. The returned function will return `True` if it is possible to reach `y` from `x` in `limit` steps or less, via only repeated applications of `f` and `g`, and `False` otherwise.

```
def make_onion(f, g):
    """
    Write a function make_onion that takes in two one-argument
    functions, F and G, and returns a function that will take in
    X, Y, and LIMIT and return True if it is possible to reach Y
    from X in LIMIT steps or less, via only repeated applications
    of F and G, and False otherwise.

    >>> add_one = lambda x: x + 1
    >>> mul_by_two = lambda y: y * 2
    >>> can_reach = make_onion(add_one, mul_by_two)
    >>> can_reach(0, 5, 4)      # 5 = add_one(mul_by_two(mul_by_two(
    add_one(0))))
    True
    >>> can_reach(0, 5, 3)      # Not possible
    False
    >>> can_reach(1, 1, 0)      # 1 = 1
    True
    >>> add_ing = lambda x: x + "ing"
    >>> add_end = lambda y: y + "end"
    >>> can_reach_string = make_onion(add_ing, add_end)
    >>> can_reach_string("cry", "crying", 1)      # "crying" =
    add_ing("cry")
    True
    >>> can_reach_string("un", "unending", 3)      # "unending" =
    add_ing(add_end("un"))
    True
    >>> can_reach_string("peach", "folding", 4)      # Not possible
    False
    """
    def can_reach(x, y, limit):
        if ____________________________________
            return ____________________________________
        elif ____________________________________
            return ____________________________________
        else:
            return ________________________ or
    ________________________
    return ____________________________________
```

Q6: Paths List

(Adapted from Fall 2013) Fill in the blanks in the implementation of `paths`, which takes as input two positive integers `x` and `y`. It returns a list of paths, where each path is a list containing steps to reach `y` from `x` by repeated incrementing or doubling. For instance, we can reach 9 from 3 by incrementing to 4, doubling to 8, then incrementing again to 9, so one path is [3, 4, 8, 9].

```
def paths(x, y):
    """Return a list of ways to reach y from x by repeated
    incrementing or doubling.
    >>> paths(3, 5)
    [[3, 4, 5]]
    >>> sorted(paths(3, 6))
    [[3, 4, 5, 6], [3, 6]]
    >>> sorted(paths(3, 9))
    [[3, 4, 5, 6, 7, 8, 9], [3, 4, 8, 9], [3, 6, 7, 8, 9]]
    >>> paths(3, 3) # No calls is a valid path
    [[3]]
    >>> paths(5, 3) # There is no valid path from x to y
    []
    """
    if ____________________________________
        return ____________________________________
    elif ____________________________________
        return ____________________________________
    else:
        a = ____________________________________
        b = ____________________________________
        return ____________________________________
```

Q7: Knapsack

You're are a thief, and your job is to pick among `n` items that are of different weights and values. You have a knapsack that supports `c` pounds, and you want to pick some subset of the items so that you maximize the value you've stolen.

Define `knapsack`, which takes a list `weights`, list `values` and a capacity `c`, and returns that max value. You may assume that item 0 weighs `weights[0]` pounds, and is worth `values[0]`; item 1 weighs `weights[1]` pounds, and is worth `values[1]`; etc.

```
def knapsack(weights, values, c):
    """
    >>> w = [2, 6, 3, 3]
    >>> v = [1, 5, 3, 3]
    >>> knapsack(w, v, 6)
    6
    """
    "*** YOUR CODE HERE ***"




















# You can use more space on the back if you want
```

Lists and Mutability

> Please refer back to Discussion 4 for an overview of lists, and Discussion 5 for an overview of mutability.

Q8: Otter Pops

Draw an environment diagram for the following program.

> Some things to remember: * When you mutate a list, you are changing the original list. * When you concatenate two lists (a + b), you are creating a new list. * When you assign a name to an existing object, you are creating another reference to that object rather than creating a copy of that object.

```
star, fish = 3, 5
otter = [1, 2, 3, 4]
soda = otter[1:]

otter[star] = fish
otter.append(soda.remove(2))
otter[otter[0]] = soda
soda[otter[0]] = otter[1]
soda = soda + [otter.pop(3)]
otter[1] = soda[1][1][0]
soda.append([soda.pop(1)])
```

You can check your solution here on PythonTutor.