INHERITANCE AND MIDTERM REVIEW 8

DATA C88C

July 16, 2024

1 Inheritance

1.1 Introduction

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 **superclass** 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* 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.

1.2 Questions

1. Assume these commands are entered in order. What would Python output?

```
class Foo:
    def __init__(self, a):
        self.a = a
    def garply(self):
        return self.baz(self.a)

class Bar(Foo):
    a = 1
    def baz(self, val):
        return val
```

```
>>> f = Foo(4)
>>> b = Bar(3)
>>> f.a
```

Solution: `4`

```
>>> b.a
```

Solution: `3`

```
>>> f.garply()
```

Solution: `AttributeError: 'Foo'object has no attribute 'baz'`

```
>>> b.garply()
```

Solution: `3`

```
>>> b.a = 9
>>> b.garply()
```

Solution: `9`

```
>>> f.baz = lambda val: val * val
>>> f.garply()
```

Solution: `16`

2 Midterm Review

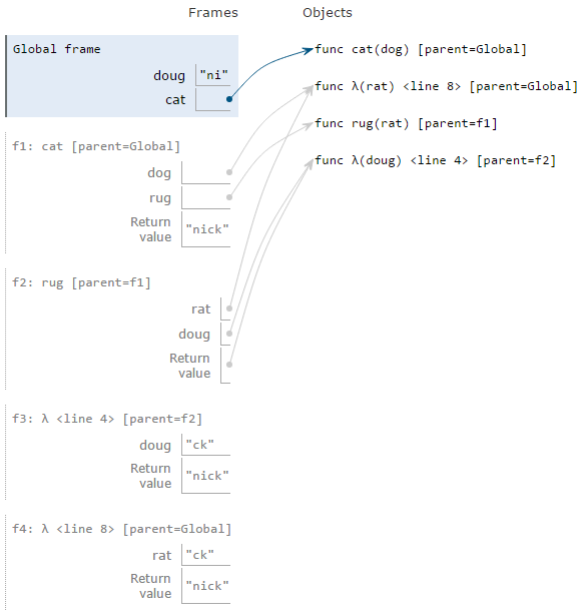
2.1 Environment Diagrams

1. Draw the environment diagram for the following code:

```
doug = "ni"
def cat(dog):
    def rug(rat):
        doug = lambda doug: rat(doug)
        return doug
    return rug(dog)("ck")

cat(lambda rat: doug + rat)
```

Solution:



Python tutor link

2.2 HOF

2. (Spring 2015) Implement the `memory` function, which takes a number `x` and a single-argument function `f`. It returns a function with a peculiar behavior that you must discover from the doctests. You may only use names and call expressions in your solution. You may not write numbers or use features of Python not yet covered in the course.

```
square = lambda x: x * x
double = lambda x: 2 * x
def memory(x, f):
    """Return a higher-order function that prints its
    memories.
    >>> f = memory(3, lambda x: x)
    >>> f = f(square)
    3
    >>> f = f(double)
    9
    >>> f = f(print)
    6
    >>> f = f(square)
    3
    None
    """
    def g(h):

        print(_____)

        return _____

    return g
```

Solution:

```
def memory(x, f):
    def g(h):
        print(f(x))
        return memory(x, h)
    return g
```

2.3 Recursion

3. (Fall 2013) The C88C staff has developed a formula for determining what a fox might say. Given three strings, a start, a middle, and an end, a fox will say the start string, followed by the middle string repeated a number of times, followed by the end string. These parts are all separated by single hyphens.

   Complete the definition of `fox_says`, which takes the three string parts of the fox's statement (start, middle, and end) and a positive integer `num` indicating how many times to repeat middle. It returns a string. You cannot use any `for` or `while` loops. Use recursion in repeat. Moreover, you cannot use string operations other than the + operator to concatenate strings together.

```
def fox_says(start, middle, end, num):
    """
    >>> fox_says('wa', 'pa', 'pow', 3)
    'wa-pa-pa-pa-pow'
    >>> fox_says('fraka', 'kaka', 'kow', 4)
    'fraka-kaka-kaka-kaka-kaka-kow'
    """
    def repeat(k):
```

Solution:

```
        if k == 1:
            return middle
        else:
            return middle + '-' + repeat(k - 1)
```

```
    return start + '-' + repeat(num) + '-' + end
```

4. Write a function that takes in a list and returns the maximum product that can be formed using nonconsecutive elements of the list. The input list will contain only numbers greater than or equal to 1.

   Hint: You can use the built-in `max` function. For example, `max(5, 3)` returns 5.

```
def max_product(lst):
    """Return the maximum product that can be formed using lst
    without using any consecutive elements
    >>> max_product([])
    1
    >>> max_product([10, 3, 1, 9, 2]) # 10 * 9
    90
    >>> max_product([5, 10, 5, 10, 5]) # 5 * 5 * 5
    125
    """
```

Solution:

```
    if lst == []:
        return 1
    else:
        return max(max_product(lst[1:]), lst[0] *
           max_product(lst[2:]))
```

At each step, we choose if we want to include the current number in our product or not:

- If we include the current number, we cannot use the adjacent number.
- If we don't use the current number, we try the adjacent number (and obviously ignore the current number).

The recursive calls represent these two alternate realities. Finally, we pick the one that gives us the largest product.