CS 61A Spring 2020

OOP, Linked Lists, Efficiency

Discussion 7: March 11, 2020 Solutions

1 Object Oriented Programming

In a previous lecture, you were introduced to the programming paradigm known as Object-Oriented Programming (OOP). OOP 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. So, a student `Angela` would be an instance of the class `Student`.

Details that all CS 61A students have, such as `name`, are called **instance attributes**. Every student has these attributes, but their values differ from student to student. An attribute that is shared among all instances of `Student` is known as a **class attribute**. An example would be the `students` attribute; the number of students that exist is not a property of any given student but rather of all of them.

All students are able to do homework, attend lecture, and go to office hours. When functions belong to a specific object, they are said to be **methods**. In this case, these actions would be bound 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 attribute:** a property of an object, specific to an instance
- **class attribute:** a property of an object, shared by all instances of a class
- **method:** an action (function) that all instances of a class may perform

Questions

1.1 Below we have defined the classes `Professor` and `Student`, implementing some of what was described above. Remember that we pass the `self` argument implicitly to instance methods when using dot-notation. There are more questions on the next page.

```
class Student:
    students = 0 # this is a class attribute
    def __init__(self, name, ta):
        self.name = name # this is an instance attribute
        self.understanding = 0
        Student.students += 1
        print("There are now", Student.students, "students")
        ta.add_student(self)

    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
```

What will the following lines output?

```
>>> snape = Professor("Snape")
>>> harry = Student("Harry", snape)
```

```
There are now 1 students
```

```
>>> harry.visit_office_hours(snape)
```

```
Thanks, Snape
```

```
>>> harry.visit_office_hours(Professor("Hagrid"))
```

```
Thanks, Hagrid
```

```
>>> harry.understanding
```

```
2
```

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

```
['Harry']
```

```
>>> x = Student("Hermione", Professor("McGonagall")).name
```

```
There are now 2 students
```

```
>>> x
```

```
'Hermione'
```

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

```
['Harry']
```

1.2 We now want to write three different classes, `Server`, `Client`, and `Email` to simulate email. Fill in the definitions below to finish the implementation! There are more methods to fill out on the next page.

We suggest that you approach this problem by first filling out the `Email` *class, then fill out the* `register_client` *method of* `Server`*, then implement the* `Client` *class, and lastly fill out the send method of the* `Server` *class.*

```
class Email:
    """Every email object has 3 instance attributes: the
    message, the sender name, and the recipient name.
    """
    def __init__(self, msg, sender_name, recipient_name):


        self.msg = msg
        self.sender_name = sender_name
        self.recipient_name = recipient_name


class Server:
    """Each Server has an instance attribute clients, which
    is a dictionary that associates client names with
    client objects.
    """
    def __init__(self):
        self.clients = {}

    def send(self, email):
        """Take an email and put it in the inbox of the client
        it is addressed to.
        """


        client = self.clients[email.recipient_name]
        client.receive(email)

    def register_client(self, client, client_name):
        """Takes a client object and client_name and adds them
        to the clients instance attribute.
        """


        self.clients[client_name] = client
```

```
class Client:
    """Every Client has instance attributes name (which is
    used for addressing emails to the client), server
    (which is used to send emails out to other clients), and
    inbox (a list of all emails the client has received).
    """
    def __init__(self, server, name):
        self.inbox = []


        self.server = server
        self.name = name
        self.server.register_client(self, self.name)

    def compose(self, msg, recipient_name):
        """Send an email with the given message msg to the
        given recipient client.
        """


        email = Email(msg, self.name, recipient_name)
        self.server.send(email)

    def receive(self, email):
        """Take an email and add it to the inbox of this
        client.
        """


        self.inbox.append(email)
```

2 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 **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.

Questions

2.1 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` to set a cat's `name` and `owner`.

```
class Cat(Pet):
    def __init__(self, name, owner, lives=9):



        Pet.__init__(self, name, owner)
        self.lives = lives


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

        >>> Cat('Thomas', 'Tammy').talk()
        Thomas says meow!
        """



        print(self.name + ' says meow!')


    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 out
        that the cat has no more lives to lose.
        """



        if self.lives > 0:
            self.lives -= 1
            if self.lives == 0:
                self.is_alive = False
        else:
            print("This cat has no more lives to lose :(")
```

Video walkthrough

2.2 More cats! Fill in this implemention of a class called `NoisyCat`, which is just like a normal `Cat`. However, `NoisyCat` talks a lot – twice as much as a regular `Cat`!

```
class ____________________: # Fill me in!
```

```
class NoisyCat(Cat):

    """A Cat that repeats things twice."""
    def __init__(self, name, owner, lives=9):
        # Is this method necessary? Why or why not?



        Cat.__init__(self, name, owner, lives)
```

No, this method is not necessary because NoisyCat already inherits Cat's __init__ method

```
    def talk(self):
        """Talks twice as much as a regular cat.

        >>> NoisyCat('Magic', 'James').talk()
        Magic says meow!
        Magic says meow!
        """



        Cat.talk(self)
        Cat.talk(self)
```

Video walkthrough

2.3 (Summer 2013 Final) What would Python display?

```
class A:
    def f(self):
        return 2
    def g(self, obj, x):
        if x == 0:
            return A.f(obj)
        return obj.f() + self.g(self, x - 1)

class B(A):
    def f(self):
        return 4
```

```
>>> x, y = A(), B()
>>> x.f()
```

2

```
>>> B.f()
```

Error (missing self argument)

```
>>> x.g(x, 1)
```

4

```
>>> y.g(x, 2)
```

8

Video walkthrough

3 Linked Lists

There are many different implementations of sequences in Python. Today, we'll explore the linked list implementation.

A linked list is either an empty linked list, or a Link object containing a `first` value and the `rest` of the linked list.

To check if a linked list is an empty linked list, compare it against the class attribute `Link.empty`:

```
if link is Link.empty:
    print('This linked list is empty!')
else:
    print('This linked list is not empty!')
```

Implementation

```
class Link:
    empty = ()
    def __init__(self, first, rest=empty):
        assert rest is Link.empty or isinstance(rest, Link)
        self.first = first
        self.rest = rest

    def __repr__(self):
        if self.rest:
            rest_str = ', ' + repr(self.rest)
        else:
            rest_str = ''
        return 'Link({0}{1})'.format(repr(self.first), rest_str)

    def __str__(self):
        string = '<'
        while self.rest is not Link.empty:
            string += str(self.first) + ' '
            self = self.rest
        return string + str(self.first) + '>'
```

Questions

3.1 Write a function that takes in a a linked list and returns the sum of all its elements. You may assume all elements in `lnk` are integers.

```
def sum_nums(lnk):
    """
    >>> a = Link(1, Link(6, Link(7)))
    >>> sum_nums(a)
    14
    """


    if lnk == Link.empty:
        return 0
    return lnk.first + sum_nums(lnk.rest)
```

3.2 Write a function that takes in a Python list of linked lists and multiplies them element-wise. It should return a new linked list.

If not all of the `Link` objects are of equal length, return a linked list whose length is that of the shortest linked list given. You may assume the `Link` objects are shallow linked lists, and that `lst_of_lnks` contains at least one linked list.

```
def multiply_lnks(lst_of_lnks):
    """
    >>> a = Link(2, Link(3, Link(5)))
    >>> b = Link(6, Link(4, Link(2)))
    >>> c = Link(4, Link(1, Link(0, Link(2))))
    >>> p = multiply_lnks([a, b, c])
    >>> p.first
    48
    >>> p.rest.first
    12
    >>> p.rest.rest.rest is Link.empty
    True
    """
```

Recursive solution:

```
    product = 1
    for lnk in lst_of_lnks:
        if lnk is Link.empty:
            return Link.empty
        product *= lnk.first
    lst_of_lnks_rests = [lnk.rest for lnk in lst_of_lnks]
    return Link(product, multiply_lnks(lst_of_lnks_rests))
```

For our base case, if we detect that any of the lists in the list of `Links` is empty, we can return the empty linked list as we're not going to multiply anything.

Otherwise, we compute the product of all the firsts in our list of `Links`. Then, the subproblem we use here is the rest of all the linked lists in our list of Links. Remember that the result of calling `multiply_lnks` will be a linked list! We'll use the product we've built so far as the first item in the returned `Link`, and then the result of the recursive call as the rest of that `Link`.

Iterative solution:

```
import operator
from functools import reduce
def prod(factors):
     return reduce(operator.mul, factors, 1)

head = Link.empty
tail = head
while Link.empty not in lst_of_lnks:
    all_prod = prod([l.first for l in lst_of_lnks])
    if head is Link.empty:
        head = Link(all_prod)
        tail = head
    else:
        tail.rest = Link(all_prod)
        tail = tail.rest
    lst_of_lnks = [l.rest for l in lst_of_lnks]
return head
```

The iterative solution is a bit more involved than the recursive solution. Instead of building the list "backwards" as in the recursive solution (because of the order that the recursive calls result in, the last item in our list will be finished first), we'll build the resulting linked list as we go along.

We use `head` and `tail` to track the front and end of the new linked list we're creating. Our stopping condition for the loop is if any of the `Links` in our list of `Links` runs out of items.

Finally, there's some special handling for the first item. We need to update both head and tail in that case. Otherwise, we just append to the end of our list using tail, and update tail.

3.3 Implement `filter_link`, which takes in a linked list `link` and a function `f` and returns a generator which yields the values of `link` for which `f` returns `True`.

Try to implement this both using a while loop and without using any form of iteration.

```
def filter_link(link, f):
    """
    >>> link = Link(1, Link(2, Link(3)))
    >>> g = filter_link(link, lambda x: x % 2 == 0)
    >>> next(g)
    2
    >>> next(g)
```

```
    StopIteration
    >>> list(filter_link(link, lambda x: x % 2 != 0))
    [1, 3]
    """

    while ___________________________:

        if ___________________________:

            ________________________

        ___________________________
```

```
def filter_link(link, f):
    while link is not Link.empty:
        if f(link.first):
            yield link.first
        link = link.rest
```

```
def filter_no_iter(link, f):
    """
    >>> link = Link(1, Link(2, Link(3)))
    >>> list(filter_no_iter(link, lambda x: x % 2 != 0))
    [1, 3]
    """

    if ____________________________:

        return

    elif ____________________________:

        ___________________________

    ____________________________
```

```
def filter_no_iter(link, f):
    if link is Link.empty:
        return
    elif f(link.first):
        yield link.first
    yield from filter_no_iter(link.rest, f)
```

1. **Midterm Review Snax**

(a) Two robots are handing at midterm snacks to 61A students who are lined up in the hallway. The left robot can hold `x` snacks at once, and the right robot can hold `y` snacks. Both robots can refill their capacity at any given time from a bottomless pit of snacks. However, when one robot (A) goes to refill snacks, the other robot (B) must wait until A returns before B continues handing out snacks. In other words, A and B must both feed a student (in full) on their respective ends of the hallway at the same time. Both robots can refill at the same time.
The list `snax` contains the number of snacks that must be given to each student in order for that student to be satisfied. Return the minimum number of refills required for both robots to feed every student in the hallway. You can assume that the individual capacity of each robot is $\geq max(snax)$, and that each robot cannot move on from its current student until the student has been satisfied.

```
def feed(snax, x, y):
    """
    >>> feed([1, 1, 1], 2, 2) # The two robots both refill once at the beginning
    2
    >>> feed([1, 2, 2], 2, 2) # Only one robot refills to feed the middle student
    3
    >>> feed([1, 1, 1, 2, 2], 2, 2)
    4
    >>> feed([3, 2, 1, 3, 2, 1, 1, 2, 3], 3, 3)
    6
    """
    def helper(lst, p, q):

        if p < 0 or q < 0:

            return float("inf")

        elif not lst:

            return 0

        elif len(lst) == 1:

            return not (p >= lst[0] or q >= lst[0])

        else:

            a = helper(lst[1:-1], p - lst[0], q - lst[-1]) # No one refills

            b = 2 + helper(lst[1:-1], x - lst[0], y - lst[-1]) # Both refill

            c = 1 + helper(lst[1:-1], x - lst[0], q - lst[-1]) # Only robot A refills

            d = 1 + helper(lst[1:-1], p - lst[0], y - lst[-1]) # Only robot B refills

            return min(a, b, c, d)

    return helper(snax, 0, 0)
```