CS 61A Summer 2019 — Object Oriented Programming

Guerrilla Section 4: July 26, 2019

1 OOP

Questions

1.1 What is the relationship between a class and an ADT?

1.2 What is the definition of a Class? What is the definition of an Instance?

1.3 What is a Class Attribute? What is an Instance Attribute?

1.4 What Would Python Display?

```
class Foo():
    x = 'bam'
    def __init__(self, x):
        self.x = x
    def baz(self):
        return self.x

class Bar(Foo):
        x = 'boom'
        def __init__(self, x):
                Foo.__init__(self, 'er' + x)
        def baz(self):
                return Bar.x + Foo.baz(self)

foo = Foo('boo')

Foo.x
|\begin{solution}
'bam'
\end{solution}|
foo.x
|\begin{solution}
```

```
'boo'
\end{solution}|
foo.baz()
|\begin{solution}
'boo'
\end{solution}|
Foo.baz()
|\begin{solution}
Error
\end{solution}|
Foo.baz(foo)
|\begin{solution}
'boo'
\end{solution}|
bar = Bar('ang')
Bar.x
|\begin{solution}
'boom'
\end{solution}|
bar.x
|\begin{solution}
'erang'
\end{solution}|
bar.baz()
|\begin{solution}
'boomerang'
\end{solution}|
```

1.5 What Would Python Display?

```
class Student:
        def __init__(self, subjects):
                self.current_units = 16
                self.subjects_to_take = subjects
                self.subjects_learned = {}
                self.partner = None

        def learn(self, subject, units):
                print('I just learned about ' + subject)
                self.subjects_learned[subject] = units
                self.current_units -= units

        def make_friends(self):
                if len(self.subjects_to_take) > 3:
                        print('Whoa! I need more help!')
                        self.partner = Student(self.subjects_to_take[1:])
                else:
                        print("I'm on my own now!")
                        self.partner = None
```

```
        def take_course(self):
                course = self.subjects_to_take.pop()
                self.learn(course, 4)
                if self.partner:
                        print('I need to switch this up!')
                        self.partner = self.partner.partner
                        if not self.partner:
                                print('I have failed to make a friend :(')

tim = Student(['Chem1A', 'Bio1B', 'CS61A', 'CS70', 'CogSci1'])
tim.make_friends()
|\begin{solution}
Whoa! I need more help!
\end{solution}|
print(tim.subjects_to_take)
|\begin{solution}
['Chem1A', 'Bio1B', 'CS61A', 'CS70', 'CogSci1']
\end{solution}|
tim.partner.make_friends()
|\begin{solution}
Whoa! I need more help!
\end{solution}|
tim.take_course()
|\begin{solution}[0.25in]
\begin{verbatim}
I just learned about CogSci1
I need to switch this up!
\end{verbatim}
\end{solution}|
tim.partner.take_course()
|\begin{solution}
I just learned about CogSci1
\end{solution}|
tim.take_course()
|\begin{solution}[0.25in]
\begin{verbatim}
I just learned about CS70
I need to switch this up!
I have failed to make a friend :(
\end{verbatim}
\end{solution}|
tim.make_friends()
|\begin{solution}
I'm on my own now!
\end{solution}|
```

1.6 Fill in the implementation for the Cat and Kitten classes. When a cat meows,

it should say "Meow, (name) is hungry" if it is hungry, and "Meow, my name is (name)" if not. Kittens do the same thing as cats, except they say "i'm baby" instead of "meow", and they say "I want mama (parents name)" after every call to meow().

```
>>>cat = Cat('Tuna')
>>>kitten = kitten('Fish', cat)
>>>cat.meow()
meow, Tuna is hungry
>>>kitten.meow()
i'm baby, Fish is hungry
I want mama Tuna
>>>cat.eat()
meow
>>>cat.meow()
meow, my name is Tuna
>>>kitten.eat()
i'm baby
>>>kitten.meow()
meow, my name is Fish
I want mama Tuna
```

```
class Cat():
     noise = 'meow'
     def __init__(self, name):
        |\begin{solution}[1in]
        \begin{verbatim}
          self.name = name
          self.hungry = True
        \end{verbatim}
         \end{solution}|
     def meow(self):
        |\begin{solution}[1in]
        \begin{verbatim}
          if self.hungry:
               print(self.noise + ', ' + self.name ' + is hungry!')
          else:
               print(self.noise + ', my name is ' + self.name)
        \end{verbatim}
         \end{solution}|
     def eat(self):
          print(self.noise)
          self.hungry = False

class Kitten(Cat):
        |\begin{solution}[1.5in]
        \begin{verbatim}
     noise = "i'm baby"
```

```
def __init__(self, name, parent):
     Cat.__init__(self, name)
     self.parent = parent
def meow(self):
     Cat.meow(self)
     print('I want mama' + parent.name)
   \end{verbatim}
    \end{solution}|
```

Check Your Understanding

1.1 Why do Foo.x and foo.x return different things?

1.2 Can we call the Foo.baz function on bar? How? What will it return?

1.3 What is tim.subjects_to_take after all the code is run?

1.4 What is the difference between a local variable, an instance variable, and a class variable? Give an example of each based on the code given.

2 Object Oriented Trees

Questions

2.1 Define **filter_tree**, which takes in a tree **t** and one argument predicate function **fn**. It should mutate the tree by removing all branches of any node where calling **fn** on its label returns **False**. In addition, if this node is not the root of the tree, it should remove that node from the tree as well.

```
def filter_tree(t, fn):
    """
    >>> t = Tree(1, [Tree(2), Tree(3, [Tree(4)]), Tree(6, [Tree(7)])])
    >>> filter_tree(t, lambda x: x % 2 != 0)
    >>> t
    tree(1, [Tree(3)])
    >>> t2 = Tree(2, [Tree(3), Tree(4), Tree(5)])
    >>> filter_tree(t2, lambda x: x != 2)
    >>> t2
    Tree(2)
    """
    |\begin{solution}[1in]
    \begin{verbatim}
    if not fn(t.label):
        t.branches = []
    else:
        for b in t.branches[:]:
            if not fn(b.label):
                t.branches.remove(b)
            else:
                filter_tree(b, fn)
    \end{verbatim}
    \end{solution}|
```

2.2 Fill in the definition for **nth_level_tree_map**, which also takes in a function and a tree, but mutates the tree by applying the function to every nth level in the tree, where the root is the 0th level.

```
def nth_level_tree_map(fn, tree, n):
   """Mutates a tree by mapping a function all the elements of a tree.
   >>> tree = Tree(1, [Tree(7, [Tree(3), Tree(4), Tree(5)]),
                Tree(2, [Tree(6), Tree(4)])])
   >>> nth_level_tree_map(lambda x: x + 1, tree, 2)
   >>> tree
   Tree(2, [Tree(7, [Tree(4), Tree(5), Tree(6)]),
            Tree(2, [Tree(7), Tree(5)])])
   """
    |\begin{solution}[1in]
    \begin{verbatim}
    def helper(tree, level):
```

```
        if level % n == 0:
            tree.label = fn(tree.label)
        for b in tree.branches:
            helper(b, level + 1)
    helper(tree, 0)
\end{verbatim}
\end{solution}|
```

Check Your Understanding

2.1 Why can we mutate trees using the Tree class? How does the Tree class differ from the Tree ADT?

2.2 How do you guarantee that your code does not recurse forever? Do we need an explicit base case?

3 Linked Lists

Questions

3.1 What is a linked list? Why do we consider it a naturally recursive structure?

3.2 Draw a box and pointer diagram for the following:

```
Link('c', Link(Link(6, Link(1, Link('a'))), Link('s')))
```

3.3 The Link class can represent lists with cycles. That is, a list may contain itself as a sublist. Implement **has_cycle** that returns whether its argument, a Link instance, contains a cycle. There are two ways to do this: iteratively with two pointers, or keeping track of Link objects we've seen already. Try to come up with both!

```
def has_cycle(link):
    """
    >>> s = Link(1, Link(2, Link(3)))
    >>> s.rest.rest.rest = s
    >>> has_cycle(s)
    True
    """
    |\begin{solution}[1in]
    \begin{verbatim}

    # solution 1
    tortoise = link
    hare = link.rest
    while tortoise.rest and hare.rest and hare.rest.rest:
        if tortoise is hare:
            return True
        tortoise = tortoise.rest
        hare = hare.rest.rest
    return False

    # solution 2
    seen = []
    while link.rest:
        if link in seen:
            return True
        seen.append(link)
        link = link.rest
    return False
    \end{verbatim}
    \end{solution}|
```

3.4 Fill in the following function, which checks to see if **sub_link**, a particular sequence of items in one linked list, can be found in another linked list (the items have to be in order, but not necessarily consecutive).

```
def seq_in_link(link, sub_link):
    """
    >>> lnk1 = Link(1, Link(2, Link(3, Link(4))))
    >>> lnk2 = Link(1, Link(3))
    >>> lnk3 = Link(4, Link(3, Link(2, Link(1))))
    >>> seq_in_link(lnk1, lnk2)
    True
    >>> seq_in_link(lnk1, lnk3)
    False
    """
    |\begin{solution}
    \begin{verbatim}
    if sub_link is Link.empty:
        return True
    if link is Link.empty:
        return False
    if link.first == sub_link.first:
        return seq_in_link(link.rest, sub_link.rest)
    else:
        return seq_in_link(link.rest, sub_link)
    \end{verbatim}
    \end{solution}|
```

Check Your Understanding

3.1 What can go in the first box of a linked list? What can go in the second?

3.2 For question 2, why do we need to store the linked list first in our code? Why can't we just iterate through it? Why can we iterate through the linked list without storing it in question 3?