MORE INTERITANCE AND LINKED LISTS 9

DATA C88C

March 18, 2024

1 Inheritance

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):
```

Solution:

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

```
    def talk(self):
        """ Print out a cat's greeting.
        >>> Cat('Thomas', 'Tammy').talk()
        Thomas says meow!
        """
```

Solution:

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

```
    def lose_life(self):
        """Decrements a cat's life by 1. When lives reaches
        zero, 'is_alive' becomes False.
```

```
        """
```

Solution:

```python
        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. 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`!

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

Solution:

```python
class NoisyCat(Cat):
```

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

Solution:

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

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

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

Solution:

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

Video walkthrough

2 Linked Lists

2.1 Introduction

The following is the `Link` class used to represent linked lists.

```
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 __getitem__(self, i):
        if i == 0:
            return self.first
        return self.rest[i-1]
    def __len__(self):
        return 1 + len(self.rest)
```

We can write `lnk.first` and `lnk.rest` to access the first element of the linked list and the rest of the linked list, respectively. In addition to the constructor `__init__`, we have the special Python methods `__getitem__` and `__len__`. Note that any method that begins and ends with two underscores is a special Python method. Special Python methods may be invoked using built-in functions and special notation. The built-in Python element selection operator, as in `lst[i]`, invokes `lst.__getitem__(i)`. Likewise, the built-in Python function `len`, as in `len(lst)`, invokes `lst.__len__()`.

However, we won't use the above special methods in the rest of this worksheet, nor in most of our linked list problems in this class. Instead, we will only use the `Link` constructor and the `self.first` and `self.rest` instance attributes. This will be an exercise in using the recursive structure of linked lists rather than treating them like regular Python lists.

For the rest of this worksheet, assume that you are only given this portion of the `Link` class 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
```

2.2 Questions

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
    """
```

Solution:

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

2. Write a iterative function `is_palindrome` that takes a LinkedList, `lnk`, and returns `True` if `lnk` is a palindrome and `False` otherwise. You can assume you have access to a `reverse` function that takes a linked list as input and returns a reversed version of the original linked list.

```
def is_palindrome(lnk):
    """
    >>> one_link = Link(1)
    >>> is_palindrome(one_link)
    True
    >>> lnk = Link(1, Link(2, Link(3, Link(2, Link(1)))))
    >>> is_palindrome(lnk)
    True
    >>> is_palindrome(Link(1, Link(2, Link(3, Link(1)))))
    False
    """
```

Solution:

```
    reversed = reverse(lnk)
    while lnk is not Link.empty and reversed.first == lnk.
       first:
        reversed = reversed.rest
        lnk = lnk.rest
    return lnk is Link.empty
```

3. Write a function that takes a sorted linked list of integers and mutates it so that all duplicates are removed.

```
def remove_duplicates(lnk):
    """
    >>> lnk = Link(1, Link(1, Link(1, Link(1, Link(5)))))
    >>> remove_duplicates(lnk)
    >>> lnk
    Link(1, Link(5))
    """
```

Solution: Recursive solution:

```
    if lnk is Link.empty or lnk.rest is Link.empty:
        return
    if lnk.first == lnk.rest.first:
        lnk.rest = lnk.rest.rest
        remove_duplicates(lnk)
    else:
        remove_duplicates(lnk.rest)
```

For a list of one or no items, there are no duplicates to remove.

Now consider two possible cases:

- If there is a duplicate of the first item, we will find that the first and second items in the list will have the same values (that is, `lnk.first == lnk.rest.first`). We can confidently state this because we were told that the input linked list is in sorted order, so duplicates are adjacent to each other. We'll remove the second item from the list.

  Finally, it's tempting to recurse on the remainder of the list (`lnk.rest`), but remember that there could still be more duplicates of the first item in the rest of the list! So we have to recurse on `lnk` instead. Remember that we have removed an item from the list, so the list is one element smaller than before. Normally, recursing on the same list wouldn't be a valid subproblem.

- Otherwise, there is no duplicate of the first item. We can safely recurse on the remainder of the list.

Iterative solution:

```
    while lnk is not Link.empty and lnk.rest is not Link.
       empty:
        if lnk.first == lnk.rest.first:
            lnk.rest = lnk.rest.rest
        else:
            lnk = lnk.rest
```

The loop condition guarantees that we have at least one item left to consider with `lnk`.

For each item in the linked list, we pause and remove all adjacent items that have the same value. Once we see that `lnk.first != lnk.rest.first`, we can safely advance to the next item. Once again, this takes advantage of the property that our input linked list is sorted.