DATA C88C
Spring 2026

Inheritance, Linked Lists
Discussion 8: March 30, 2026

Inheritance

To avoid redefining attributes and methods for similar classes, we can write a single **base class** from which more specialized classes **inherit**. For example, we can write a class called `Pet` and define `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):
        super().talk()
        print('This Dog says woof!')
```

Inheritance represents a hierarchical relationship between two or more classes where one class **is a** more specific version of the other: a dog **is a** pet. (We use "**is a**" to describe this sort of relationship in OOP languages, not to refer to the Python `is` operator.)

Since `Dog` inherits from `Pet`, the `Dog` class will also inherit the `Pet` class's methods, so we don't have to redefine `__init__` or `eat`. We do want each `Dog` to `talk` in a `Dog`-specific way, so we can **override** the `talk` method.

We can use `super()` to refer to the superclass of `self`, and access any superclass methods as if we were an instance of the superclass. For example, `super().talk()` in the `Dog` class will call the `talk` method from the `Pet` class, but passes in the `Dog` instance as the `self`.

Q1: Cat

Below is the implementation of a `Pet` class. Each pet has two instance attributes (`name` and `owner`), as well as one instance method (`talk`).

```
class Pet:

    def __init__(self, name, owner):
        self.name = name
        self.owner = owner

    def talk(self):
        print(self.name)
```

Implement the `Cat` class, which inherits from the `Pet` class seen above. To complete the implementation, override or implement the following methods:

`___init___`

Set the `Cat`'s `name` and `owner` attributes, and also add 2 new attributes:

1. `is_hungry` - should be set to `False`
2. `fullness` - should be set to whatever the `fullness` parameter is

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

`talk`

Print out a cat's greeting, which is `"<name of cat> says meow!"`.

`get_hungry`

Decrements a cat's `fullness` level by 1. When `fullness` reaches zero, `is_hungry` becomes `True`. If this is called after `fullness` has already reached zero, print the message `"<name of cat> is hungry."`

`eat`

This method is called when the cat eats some food.

If the cat is hungry, after calling this method both of the following should be true:

1. The cat's `fullness` value should be set to whatever `Cat.default_fullness` is.
2. The cat's `is_hungry` value should be `False`.

Also print out the food the cat ate. For example, if a cat named Thomas ate fish, print out `'Thomas ate a fish!'`

Otherwise, if the cat wasn't hungry, print `'<name of cat> is not hungry.'`

```
class Cat(Pet):
    default_fullness = 5

    def __init__(self, name, owner, fullness=default_fullness):
        """
        >>> cat = Cat('Thomas', 'Tammy')
        >>> cat.name
        'Thomas'
        >>> cat.owner
        'Tammy'
        >>> cat.fullness # use default fullness value
        5
        >>> cat.is_hungry
        False
        >>> cat2 = Cat('Meow Meow', 'Yoobin', 3)
        >>> cat2.fullness # use fullness value that was passed in
        3
        """
        super().__init__(name, owner)
        self.fullness = fullness
        self.is_hungry = False

    def talk(self):
        """
        >>> Cat('Thomas', 'Tammy').talk()
        Thomas says meow!
        >>> Cat('Meow Meow', 'ThuyAnh').talk()
        Meow Meow says meow!
        """
        print(self.name + ' says meow!')

    def get_hungry(self):
        """
        >>> cat = Cat('Thomas', 'Tammy', 2)
        >>> cat.is_hungry
        False
        >>> cat.fullness
        2
        >>> cat.get_hungry()
        >>> cat.is_hungry
        False
        >>> cat.fullness
        1
        >>> cat.get_hungry()
        >>> cat.is_hungry
        True
        >>> cat.fullness
        0
        >>> cat.get_hungry()
        Thomas is hungry.
        >>> cat.is_hungry
        True
```

Linked Lists

A linked list is a `Link` object or `Link.empty`.

You can mutate a `Link` object `s` in two ways: - Change the first element with `s.first = ...` - Change the rest of the elements with `s.rest = ...`

You can make a new `Link` object by calling `Link`: - `Link(4)` makes a linked list of length 1 containing 4. - `Link(4, s)` makes a linked list that starts with 4 followed by the elements of linked list `s`.

```
class Link:
    """A linked list is either a Link object or Link.empty

    >>> s = Link(3, Link(4, Link(5)))
    >>> s.rest
    Link(4, Link(5))
    >>> s.rest.rest.rest is Link.empty
    True
    >>> s.rest.first * 2
    8
    >>> print(s)
    (3 4 5)
    """
    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_repr = ', ' + repr(self.rest)
        else:
            rest_repr = ''
        return 'Link(' + repr(self.first) + rest_repr + ')'

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

Q2: Sum Two Ways

Implement both `sum_rec` and `sum_iter`. Each one takes a linked list of numbers `s` and a non-negative integer `k` and returns the sum of the first `k` elements of `s`. If there are fewer than `k` elements in `s`, all of them are summed. If `k` is 0 or `s` is empty, the sum is 0.

Use recursion to implement `sum_rec`. Don't use recursion to implement `sum_iter`; use a `while` loop instead.

To get started on the recursive implementation, consider the example `a = Link(1, Link(6, Link(8)))`, and the call `sum_rec(a, 2)`. Write down the recursive call to `sum_rec` that would help compute `sum_rec(a, 2)`. Then, write down what that recursive call should return. Discuss how this return value is useful in computing the return value of `sum_rec(a, 2)`.

```
def sum_rec(s, k):
    """Return the sum of the first k elements in s.

    >>> a = Link(1, Link(6, Link(8)))
    >>> sum_rec(a, 2)
    7
    >>> sum_rec(a, 5)
    15
    >>> sum_rec(Link.empty, 1)
    0
    """
    # Use a recursive call to sum_rec; don't call sum_iter
    if k == 0 or s is Link.empty:
        return 0
    return s.first + sum_rec(s.rest, k - 1)

def sum_iter(s, k):
    """Return the sum of the first k elements in s.

    >>> a = Link(1, Link(6, Link(8)))
    >>> sum_iter(a, 2)
    7
    >>> sum_iter(a, 5)
    15
    >>> sum_iter(Link.empty, 1)
    0
    """
    # Don't call sum_rec or sum_iter
    total = 0
    while k > 0 and s is not Link.empty:
        total, s, k = total + s.first, s.rest, k - 1
    return total
```

Discussion time: When adding up numbers, the intermediate sums depend on the order. `(1 + 3) + 5` and `1 + (3 + 5)` both equal 9, but the first one makes 4 along the way while the second makes 8 along the way. For the same linked list `s` and length `k`, will `sum_rec` and `sum_iter` both make the same intermediate sums along the way?

Q3: Duplicate Link

Write a function `duplicate_link` that takes in a linked list `s` and a `value`. `duplicate_link` will mutate `s` such that if there is a linked list node that has a `first` equal to `value`, that node will be duplicated. Note that you should be **mutating the original linked list** `s`; you will need to create new `Link`s, but you should not be returning a new linked list.

> **Note**: In order to insert a link into a linked list, you need to modify the `.rest` of certain links. We encourage you to draw out a doctest to visualize!

```
def duplicate_link(s: Link, val: int) -> None:
    """Mutates s so that each element equal to val is followed by another val.

    >>> x = Link(5, Link(4, Link(5)))
    >>> duplicate_link(x, 5)
    >>> x
    Link(5, Link(5, Link(4, Link(5, Link(5)))))
    >>> y = Link(2, Link(4, Link(6, Link(8))))
    >>> duplicate_link(y, 10)
    >>> y
    Link(2, Link(4, Link(6, Link(8))))
    >>> z = Link(1, Link(2, Link(2, Link(3))))
    >>> duplicate_link(z, 2) # ensures that back to back links with val are both
    duplicated
    >>> z
    Link(1, Link(2, Link(2, Link(2, Link(2, Link(3))))))
    """
    if s is Link.empty:
        return
    elif s.first == val:
        remaining = s.rest
        s.rest = Link(val, remaining)
        duplicate_link(remaining, val)
    else:
        duplicate_link(s.rest, val)
```