Inheritance and String Representation

Announcements

Attribute Lookup Practice

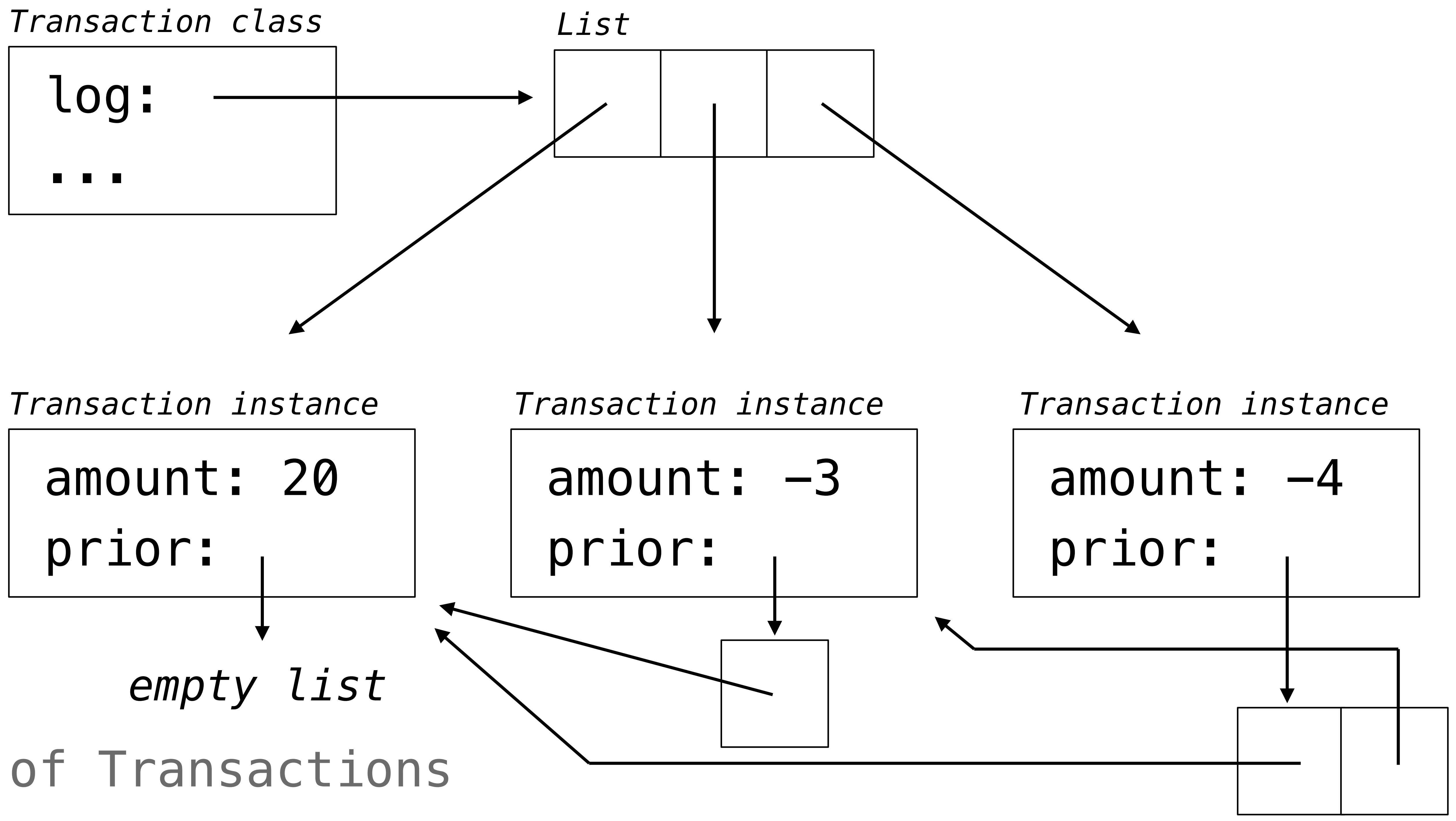
Class Attributes

A class attribute can be accessed from either an instance or its class. There is only one value for a class attribute, regardless of how many instances.

```
class Transaction:
    """A logged transaction.

    >>> s = [20, -3, -4]
    >>> ts = [Transaction(x) for x in s]
    >>> ts[1].balance()
    17
    >>> ts[2].balance()
    13
    """
    log = []

    def __init__(self, amount):
        self.amount = amount
        self.prior = list(self.log) # a list of Transactions
        self.log.append(self)

    def balance(self):
        """The sum of amounts for this transaction and all prior transactions"""
        return self.amount + sum( [t.amount for t in self.prior] )
```

Always bound to some Transaction instance

Equivalently: list(type(self).log) **or** list(Transaction.log)



Accessing Attributes

Using getattr, we can look up an attribute using a string

```
>>> tom_account.balance
10
```

```
>>> getattr(tom_account, 'balance')
10

>>> hasattr(tom_account, 'deposit')
True
```

getattr and dot expressions look up a name in the same way

Looking up an attribute name in an object may return:

- One of its instance attributes, or
- One of the attributes of its class

(Demo)

Example: Close Friends

```
class Friend:
    def __init__(self, name):
        self.name = name
        self.heard_from = {}

    def hear_from(self, friend):
        if friend not in self.heard_from:
            self.heard_from[friend] = 0
        self.heard_from[friend] += 1
        friend.just_messaged = self

   def how_close(self, friend):
        bonus = 0

         hasattr(self, 'just_messaged')       self.just_messaged is friend
        if ______________________________ and ______________________________:
            bonus = 3

                friend.heard_from.get(self, 0)
        return ______________________________ + bonus

    def closest(self, friends):

                                self.how_close
        return max(friends, key=______________________________)
```

A **Friend** instance tracks the number of times they **hear_from** each other friend.

A **Friend just_messaged** the friend that most recently heard from them.

how_close is one Friend (**self**) to another (**friend**)?

- The number of times **friend** has heard from **self**
- Plus a bonus of 3 if they are the one that most recently heard from **self**

self's closest friend among a list of **friends** is the one with the largest **self.how_close(friend)** value

Inheritance

Inheritance Example

A CheckingAccount is a specialized type of Account

```
>>> ch = CheckingAccount('Tom')
>>> ch.interest     # Lower interest rate for checking accounts
0.01
>>> ch.deposit(20)  # Deposits are the same
20
>>> ch.withdraw(5)  # Withdrawals incur a $1 fee
14
```

Most behavior is shared with the base class Account

```
class CheckingAccount(Account):
    """A bank account that charges for withdrawals."""
    withdraw_fee = 1
    interest = 0.01
    def withdraw(self, amount):
        return Account.withdraw(self, amount + self.withdraw_fee)
                                      or
        return super().withdraw(      amount + self.withdraw_fee)
```

Looking Up Attribute Names on Classes

Base class attributes *aren't* copied into subclasses!

To look up a name in a class:

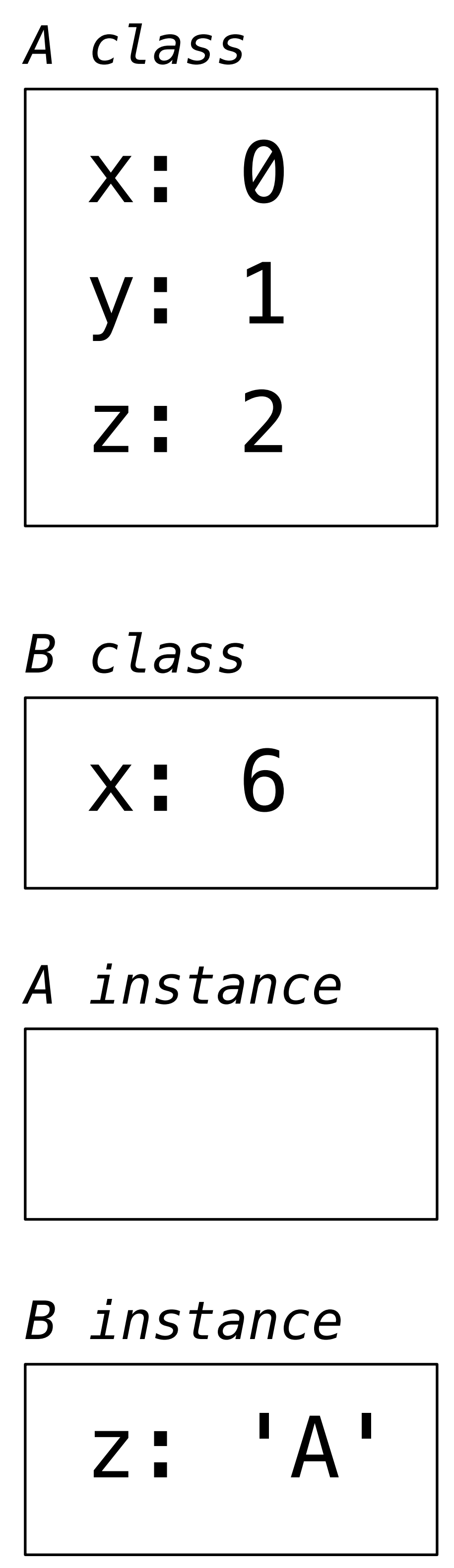
1. If it names an attribute in the class, return the attribute value.

2. Otherwise, look up the name in the base class, if there is one.

```
>>> ch = CheckingAccount('Tom')  # Calls Account.__init__
>>> ch.interest     # Found in CheckingAccount
0.01
>>> ch.deposit(20)  # Found in Account
20
>>> ch.withdraw(5)  # Found in CheckingAccount
14
```

Example: Three Attributes

```
class A:
    x, y, z = 0, 1, 2

    def f(self):
        return [self.x, self.y, self.z]

class B(A):
    """What would Python Do?

    >>> A().f()

    [0, 1, 2]

    >>> B().f()
     [6, 1, 'A']
    _____________

    """
    x = 6
    def __init__(self):
        self.z = 'A'
```



Break: 5 minutes

String Representations

String Representations

In Python, all objects produce two string representations:

- The **str** is (often) legible to **humans** & shows up when you **print**
- The **repr** is (often) legible to **Python** & shows up when you **evaluate** interactively

The **str** and **repr** strings are often the same, but not always

```
>>> from fractions import Fraction
>>> half = Fraction(1, 2)
>>> str(half)
'1/2'
>>> repr(half)
'Fraction(1, 2)'
>>> print(half)
1/2
>>> half
Fraction(1, 2)
```

If a type only defines a repr string, then the repr string is also the str string.

(Demo)

Class Practice

Spring 2023 Midterm 2 Question 2(a)

```
class Letter:
    def __init__(self, contents):

        self.contents = contents

        self.sent = False
        _________________

    def send(self):

        if self.sent:

            print(self, 'was already sent.')

        else:
            print(self, 'has been sent.')

            self.sent = True
            ________________

            return Letter(self.contents.upper())
                   _____________________________

    def __repr__(self):
        return self.contents
```

Implement the **Letter** class. A **Letter** has two instance attributes: **contents** (a **str**) and **sent** (a **bool**). Each **Letter** can only be sent once. The **send** method prints whether the letter was sent, and if it was, returns the reply, which is a new **Letter** instance with the same contents, but in all caps.
Hint: 'hi'.upper() evaluates to 'HI'.

```
"""A letter receives an all-caps reply.

>>> hi = Letter('Hello, World!')
>>> hi.send()
Hello, World! has been sent.
HELLO, WORLD!
>>> hi.send()
Hello, World! was already sent.
>>> Letter('Hey').send().send()
Hey has been sent.
HEY has been sent.
HEY
"""
```

Spring 2023 Midterm 2 Question 2(b)

```
class Numbered(Letter):

    number = 0

    def __init__(self, contents):

        super().__init__(contents)

        self.number = Numbered.number
        ____________________________

        Numbered.number += 1
        ____________________________

    def __repr__(self):

        return '#' + str(self.number)
                     ____________________
```

Implement the **Numbered** class. A **Numbered** letter has a **number** attribute equal to how many numbered letters have previously been constructed. This **number** appears in its **repr** string. Assume **Letter** is implemented correctly.

```
"""A numbered letter has a different
repr method that shows its number.

>>> hey = Numbered('Hello, World!')
>>> hey.send()
#0 has been sent.
HELLO, WORLD!
>>> Numbered('Hi!').send()
#1 has been sent.
HI!
>>> hey
#0
"""
```