Review 1

Call Frames; Diagramming Objects

The Big Issue

- Cannot answer questions on this topic without
  - drawing variables
  - drawing frames for function calls
  - drawing objects when they are created
- Learning to do this is useful in general
  - Helps you “think like a computer”
  - Easier to find errors in your programs.

What Do You Need to Know?

- Major topics
  - *local variables (in a function body)*
  - *function call (call frames, call stack)*
  - *constructor call (in addition to call frames)*

- Examples from previous exams
  - Question 3 on prelim 1
  - Question 5 on prelim 2

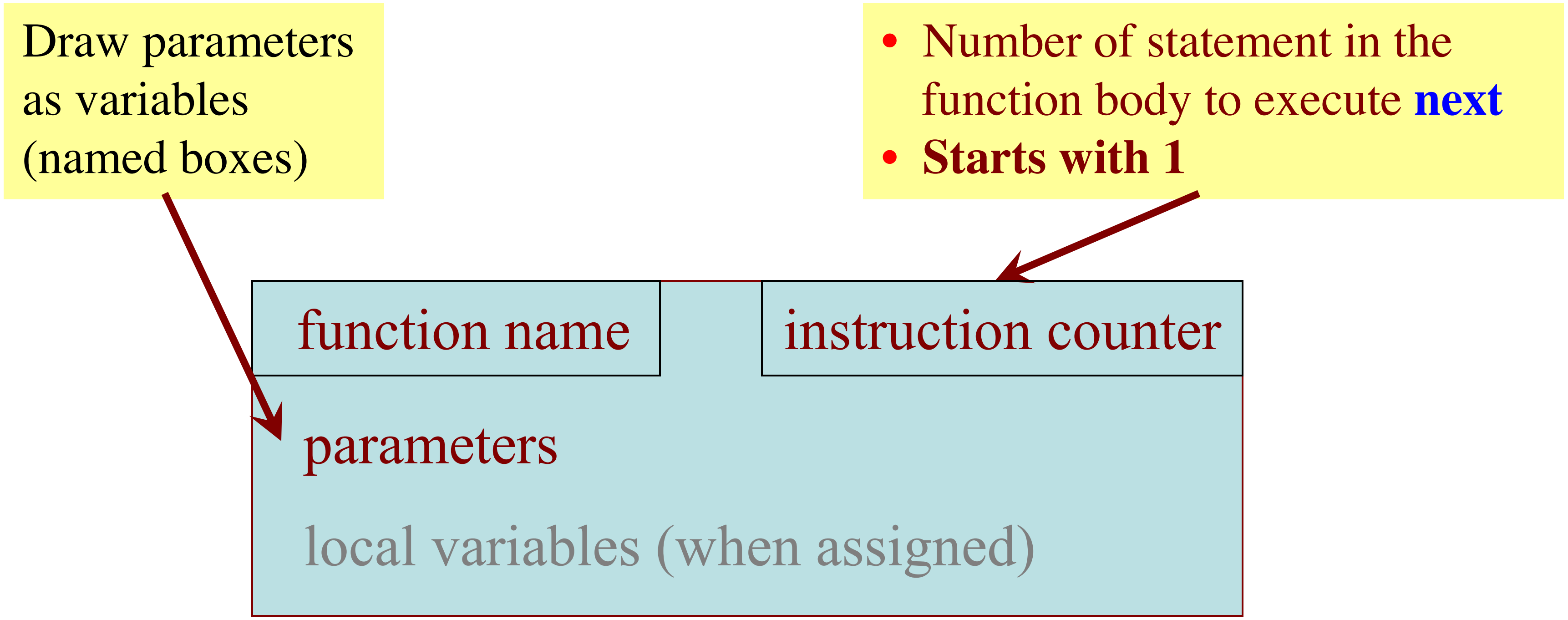
Important

- Code execution is an important part of the final
- You need to know how to
  - draw variables
  - draw call frames
  - draw objects

The purpose of such questions on executing statements with constructs and function calls is to test your understanding of how Python programs are executed

The Frame (box) for a Function Call

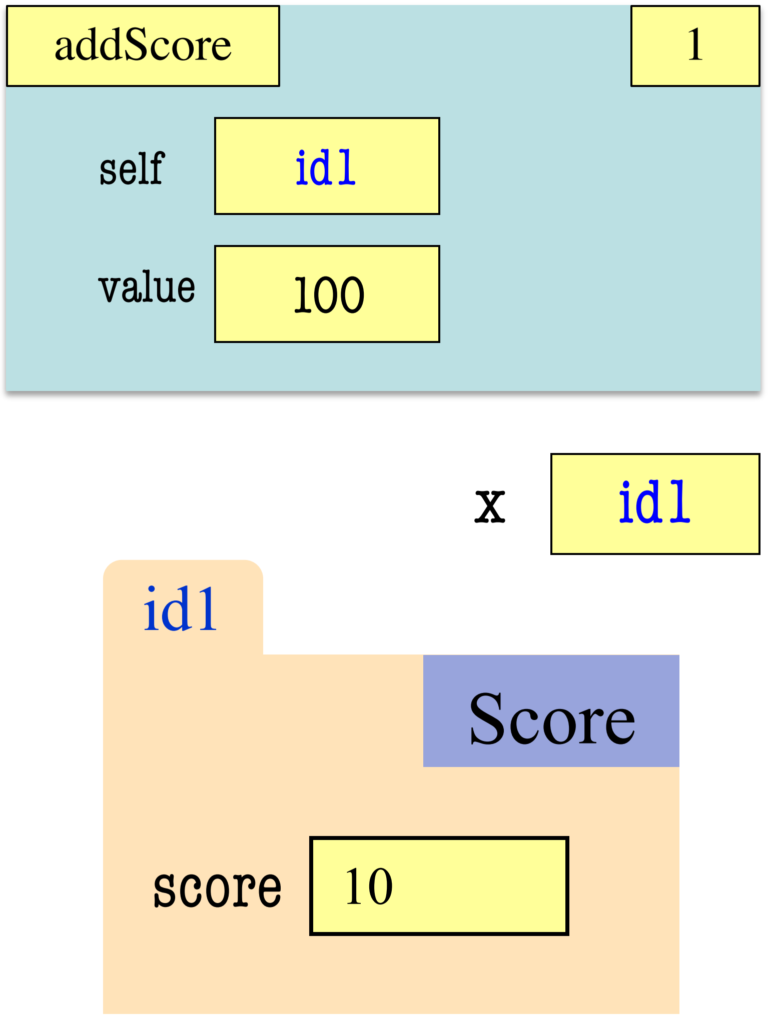
- **Function Frame**: Representation of function call
- A **conceptual model** of Python


Draw parameters as variables (named boxes)
Number of statement in the function body to execute next
Starts with 1
function name
instruction counter
parameters
local variables (when assigned)

To Execute the Method: x.addScore(100)

1. Draw a frame for the call
2. Assign the arguments to the parameters (in frame)
3. Execute the method body
   - Look for variables in frame
   - If an attribute, follow the name into Heap Space
4. Erase the frame

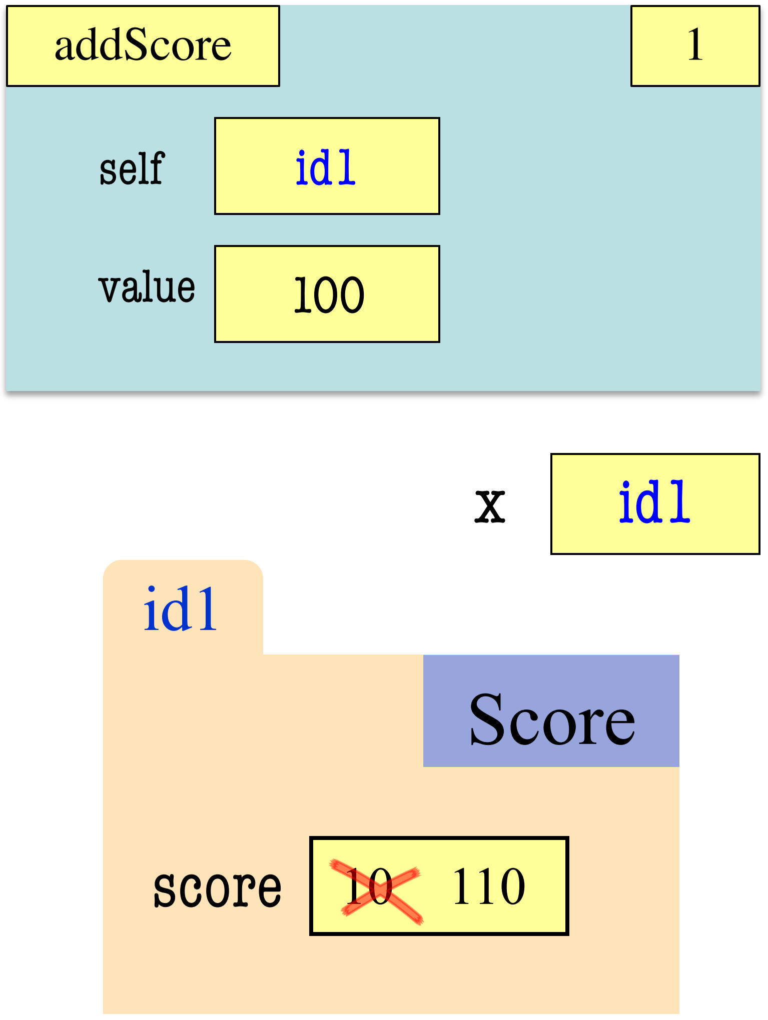
```
class Score(object):
    …
    def addScore(self,value):
        """Add value to score attr"""
        self._score = self._score+value
```



To Execute the Method: x.addScore(100)

1. Draw a frame for the call
2. Assign the arguments to the parameters (in frame)
3. Execute the method body
   - Look for variables in frame
   - If an attribute, follow the name into Heap Space
4. Erase the frame

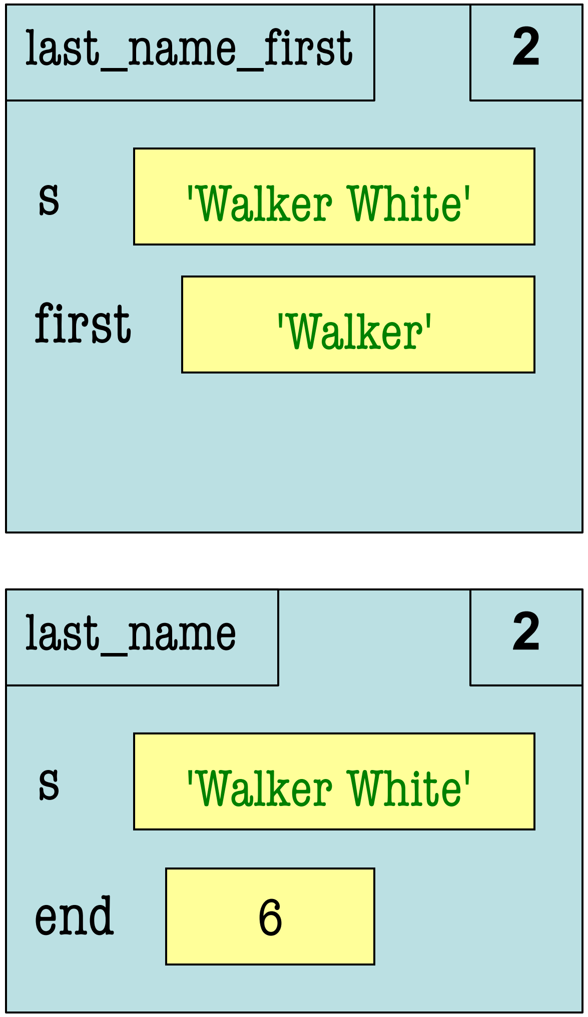
```
class Score(object):
    ...
    def addScore(self,value):
        """Add value to score attr"""
        self._score = self._score+value
```



Call Stacks: Given a Line to Reach

```
  def last_name_first(s):
      """Precondition: s in the form
      <first-name> <last-name>"""
1     first = first_name(s)
2     last = last_name(s)
3     return last + '.' + first

  def last_name(s):
      """Prec: see last_name_first"""
1     end = s.find(' ')
2     return s[end+1:]
```

Execute to here



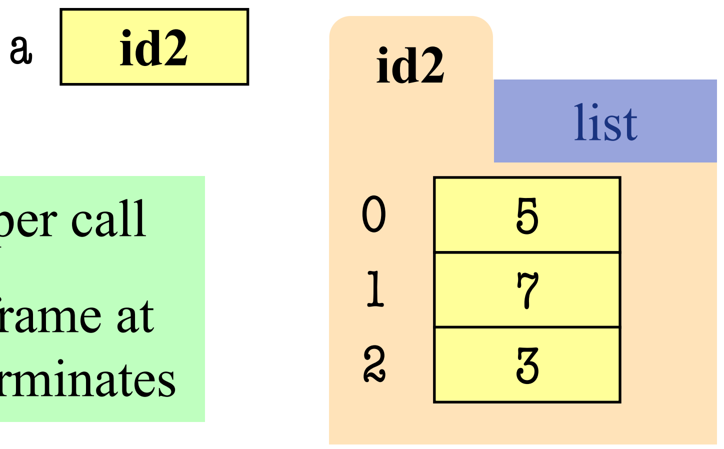
(Modified) Question from Previous Years

```
def reverse(b):
    """Reverse elements of b in place
    (does not make a copy)
    Pre: b is a list"""
1   reverse_part(b,0,len(b)-1)

def reverse_part(b,h,k):
    """Reverse b[h..k] in place
    Pre: b is a list; h, k are in b"""
1   if h >= k:
2       return
3   temp = b[h]
4   b[h] = b[k]
5   b[k] = temp
6   reverse_part(b,h+1,k-1)
```

- Execute the call
  - a = [5,7,3]; reverse(a)
  - Use 'folder' for list a below
  - Stop upon completing line 2
  - **Draw call frame at that time**!

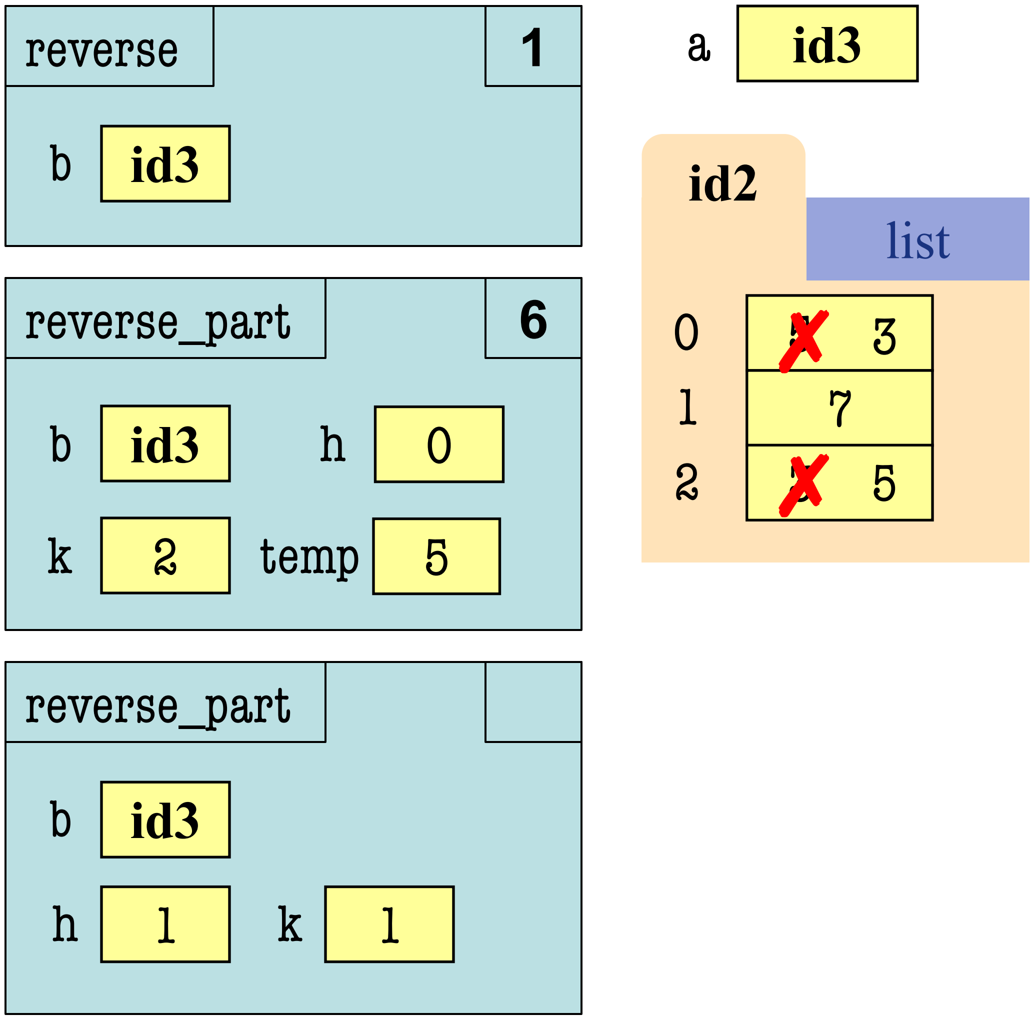
Give only one frame per call

Give the state of the frame at last line before call terminates



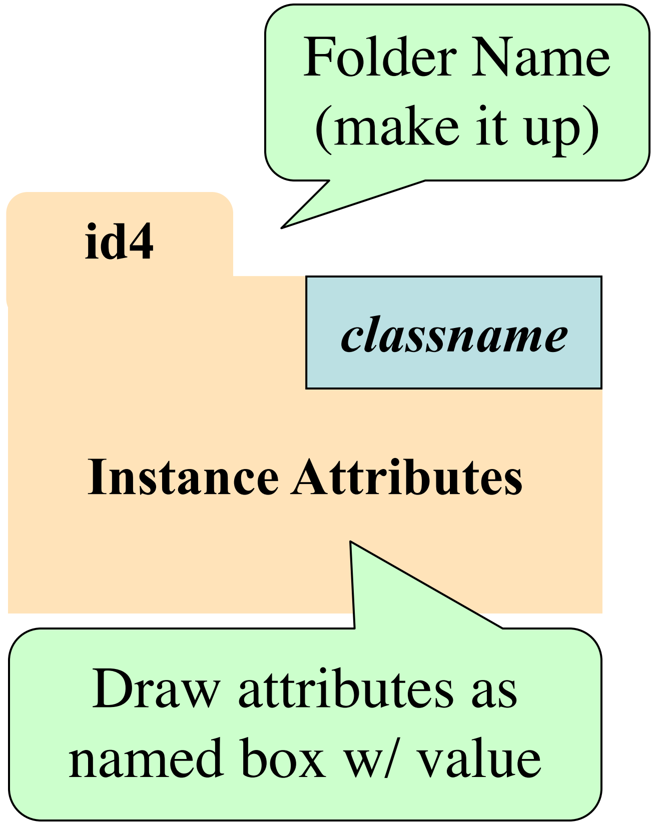
Execute the Call reverse([5,7,3]) to Line 2

```
def reverse(b):
    """Reverse elements of b in place
    (does not make a copy)
    Pre: b is a list"""
1   reverse_part(b,0,len(b)-1)

def reverse_part(b,h,k):
    """Reverse b[h..k] in place
    Pre: b is a list; h, k are in b"""
1   if h >= k:
2       return
3   temp = b[h]
4   b[h] = b[k]
5   b[k] = temp
6   reverse_part(b,h+1,k-1)
```

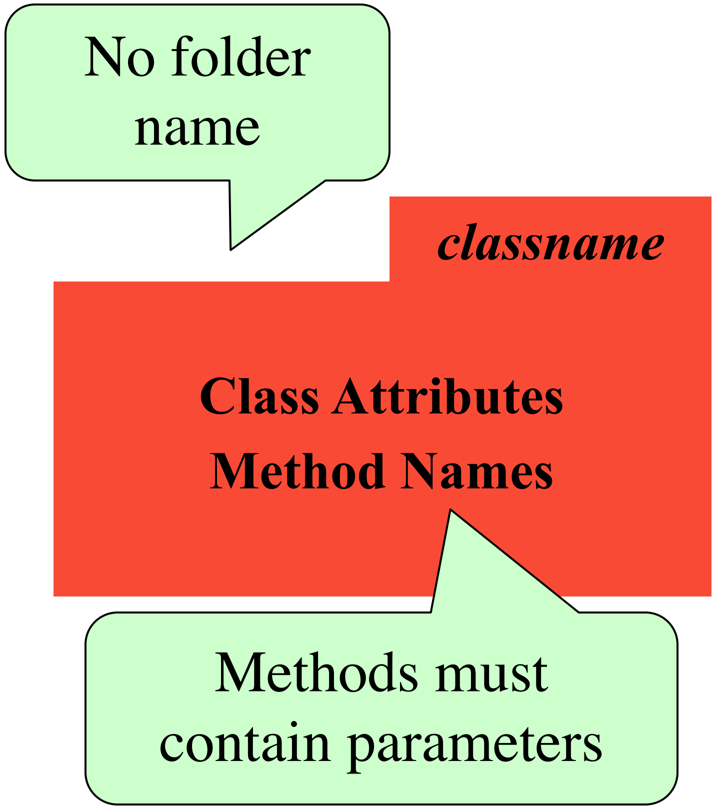


Diagramming Objects (Folders)

Object Folder

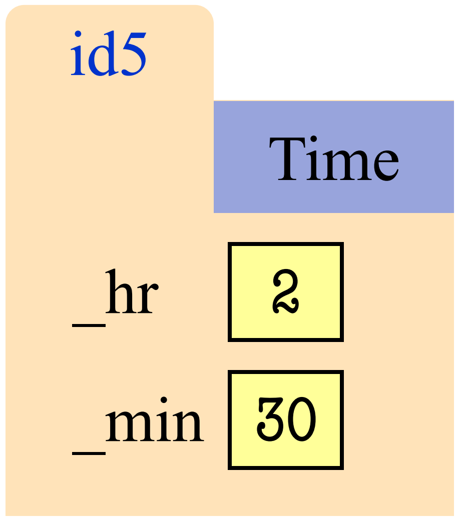


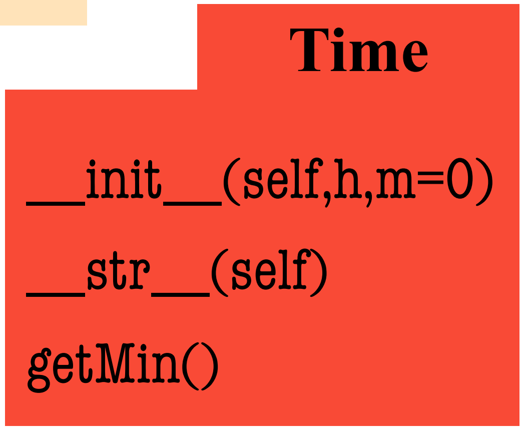
Class Folder



Diagramming Example

```
class Time(object):
    """Instance attributes:
        _hr: hour of day [int, 0..23]
        _min: minute of hour [int, 0..59]"""

    def getMin(self):
        """Return: minute of hour"""
        return self._min

    def __init__(self, h, m=0):
        """Initializer: new time h:m"""
        self._hr = h;  self._min = m

    def __str__(self):
        """Returns string '<hr>:<min>' """
        return `self._hr` + ':' + `self._min`
```





Evaluation of a Constructor Call

3 steps to evaluating the call C(args)

- ***Create a new folder*** (object) of class C
  - Give it with a unique id (any number will do)
  - Folder goes into the heap
- Execute the ***method*** __init__(args)
- Return ***the identifier*** of the object as ***the value***
  - A constructor call is an *expression*, not a command
  - Does not put name in a variable unless you **assign it**

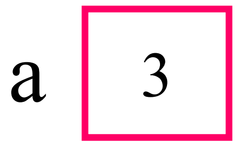
Code Segment (with Constructors)

```
class C(object):
    f = 0
    def __init__(self, k):
        self.f = k
```

```
a = 3
x = C(a) # C a class
y = C(a)
x = y
```

Code Segment (with Constructors)

```python
class C(object):
    f = 0
    def __init__(self, k):
        self.f = k
```

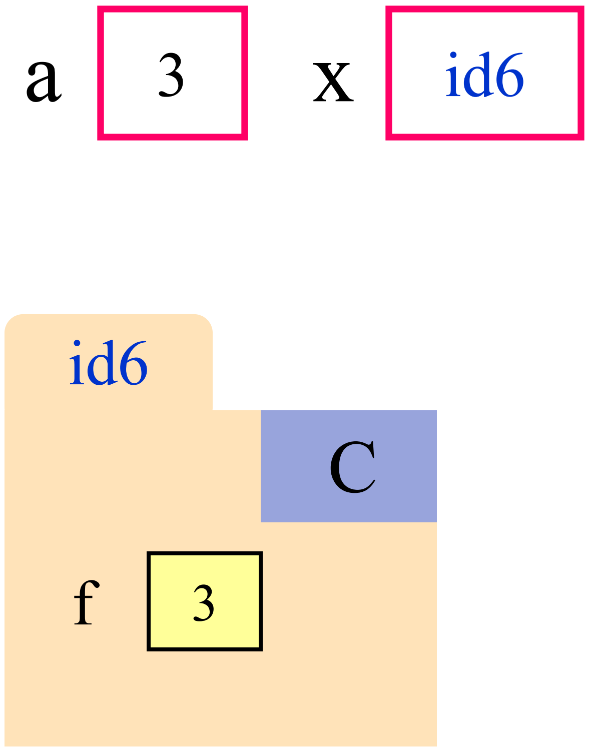


```python
a = 3
x = C(a) # C a class
y = C(a)
x = y
```

Code Segment (with Constructors)

```
class C(object):
    f = 0
    def __init__(self, k):
        self.f = k
```

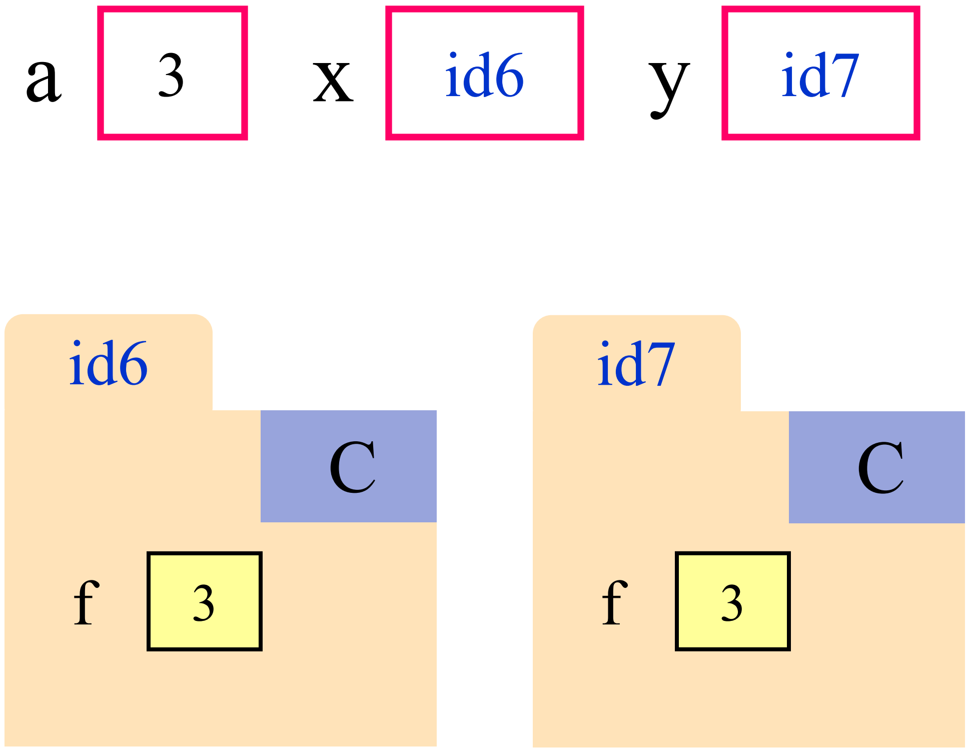
```
a = 3
x = C(a) # C a class
y = C(a)
x = y
```



Code Segment (with Constructors)

```
class C(object):
    f = 0
    def __init__(self, k):
        self.f = k
```

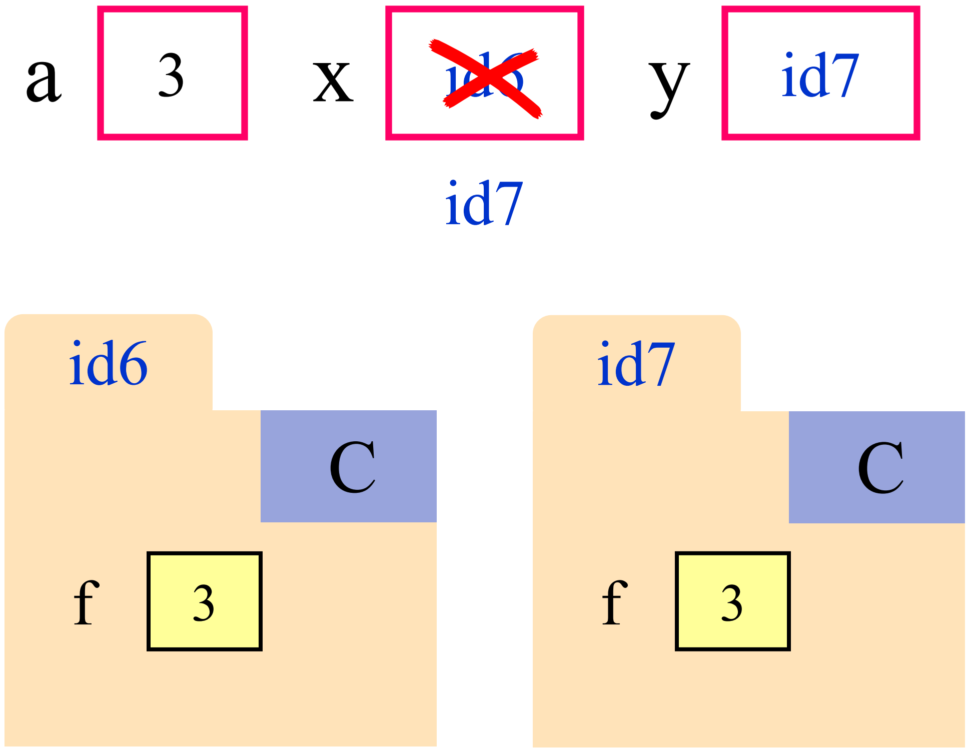
```
a = 3
x = C(a) # C a class
y = C(a)
x = y
```



Code Segment (with Constructors)

```
class C(object):
    f = 0
    def __init__(self, k):
        self.f = k
```

```
a = 3
x = C(a) # C a class
y = C(a)
x = y
```



old reference is lost

Code Execution (Q4 from 2008 fall final, modified)

Execute the call: session()

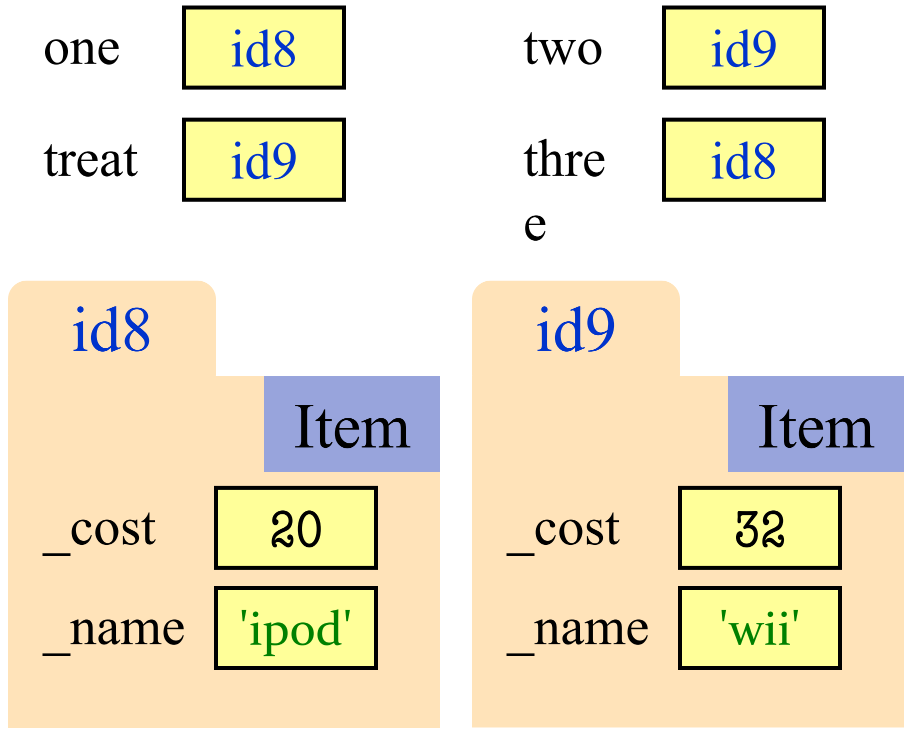
```
def session()
    one = Item('ipod', 20)
    two = Item('wii', 32)
    treat = two
    three = one
    three.add(4)
    print one
    print 'Cost of item one: '+str(one.getCost())
    print ('Are they the same? ' +
              str(one.getName()==two.getName()))
    print ('Are they the same? ' +
              str(one.getName()==treat.getName()))
    print ('Are they the same? ' +
              str(one.getName()==three.getName()))
```

```
class Item(object):
    """Instance attributes:
        _cost: cost of this item [float > 0]
        _name: item name [nonempty str]
    """

    def __init__(self, t, c):
        """Initializer: new Item with name t, cost c"""
        self._name = t;  self._cost = c

    def getCost(self):
        """Return: cost of this item """
        return self._cost

    def getName(self):
        """Return: item's name"""
        return self._name

    def __str__(self):
        """Returns '<name>:<cost>' as representation"""
        return self.name + ':' + str(self.cost)

    def add(self, d):
        """Add d to this item's cost"""
        self._cost = self._cost + d
```

Code Execution (Q4 from 2008 fall final, modified)

Execute the call: session()

```
def session()
    one = Item('ipod', 20)
    two = Item('wii', 32)
    treat = two
    three = one
    three.add(4)
    print one
    print 'Cost of item one: '+str(one.getCost())
    print ('Are they the same? ' +
              str(one.getName()==two.getName()))
    print ('Are they the same? ' +
              str(one.getName()==treat.getName()))
    print ('Are they the same? ' +
              str(one.getName()==three.getName()))
```



Code Execution (Q4 from 2008 fall final, modified)

Execute the call: session()

```
   def session()
1      one = Item('ipod', 20)
2      two = Item('wii', 32)
3      treat = two
4      three = one
5      three.add(4)
6      print one
7      print 'Cost of item one: '+str(one.getCost())
8      print ('Are they the same? ' +
               str(one.getName()==two.getName()))
9      print ('Are they the same? ' +
               str(one.getName()==treat.getName()))
10     print ('Are they the same? ' +
               str(one.getName()==three.getName()))
```

one	id8	two	id9
treat	id9	thre e	id8

Output:

6 : 'ipod:24'

7 : 'Cost of item one: 24'

8 : 'Are they the same? False'

9 : 'Are they the same? False'

10 : 'Are they the same? True'

Example from Fall 2019(!)

```
class Cornellian(object):
    # Instance attributes:
    # _cuid: Cornell id, an int > 0
    # _name: full name, a nonempty str

    NEXT = 1 # Class Attribute

    def getName(self):
        """"Returns the name."""
        return self._name
    …
    def __init__(self, n):
        """Initializes Cornellian with name n."""
        self._name = n
        self._cuid = Cornellian.NEXT
        Cornellian.NEXT = Cornellian.NEXT+1
    …
```

Execute:

```
>>> a = Cornellian('Alice')
>>> b = Cornellian('Bob')
```

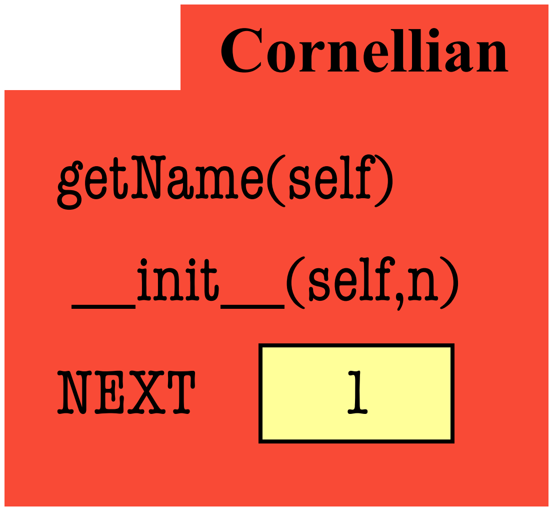
Pay close attention to class attribute NEXT

Example from Fall 2019(!)

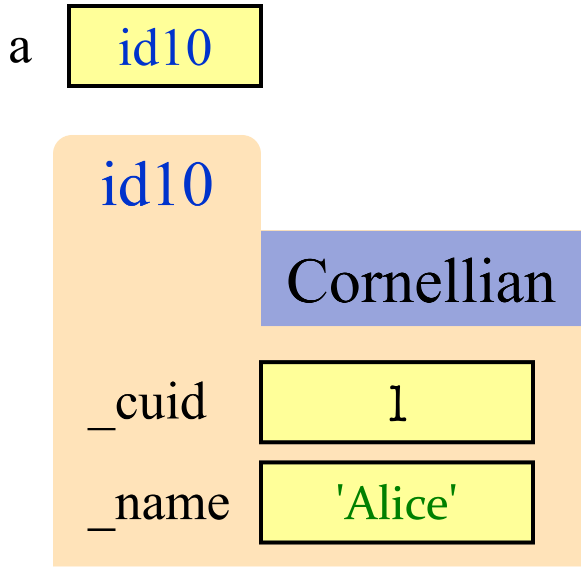
```
class Cornellian(object):
    # Instance attributes:
    # _cuid: Cornell id, an int > 0
    # _name: full name, a nonempty str

    NEXT = 1 # Class Attribute

    def getName(self):
        """Returns the name."""
        return self._name
    ...
    def __init__(self, n):
        """Initializes Cornellian with name n."""
        self._name = n
        self._cuid = Cornellian.NEXT
        Cornellian.NEXT = Cornellian.NEXT+1
    ...
```

Execute:

```
>>> a = Cornellian('Alice')
>>> b = Cornellian('Bob')
```

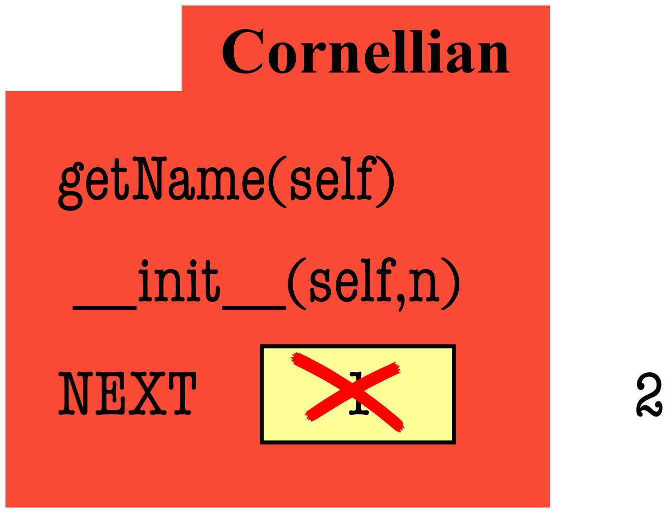


Example from Fall 2019(!)

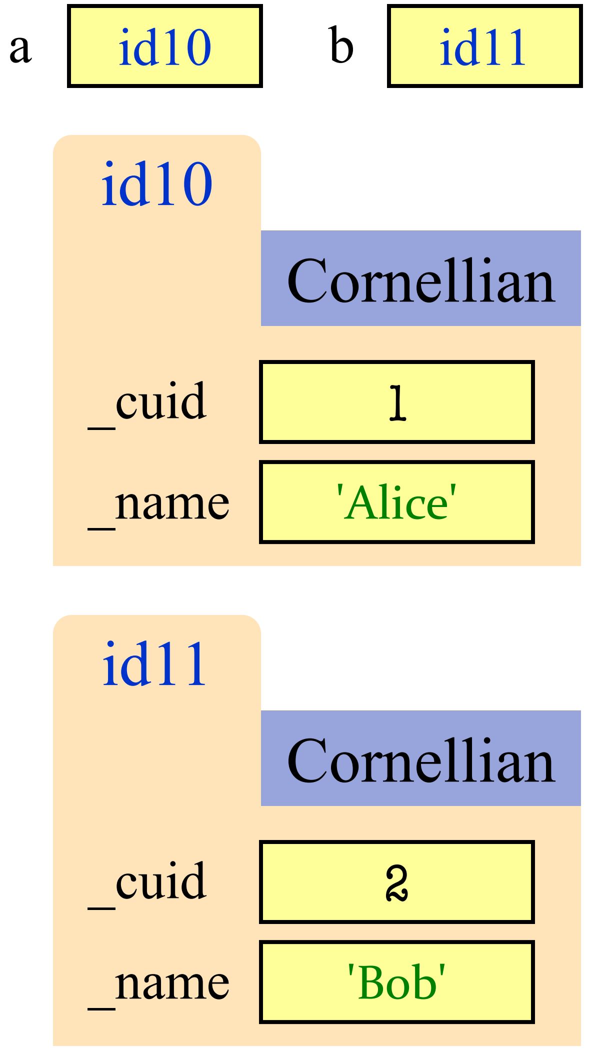


Execute:

```
>>> a = Cornellian('Alice')
>>> b = Cornellian('Bob')
```

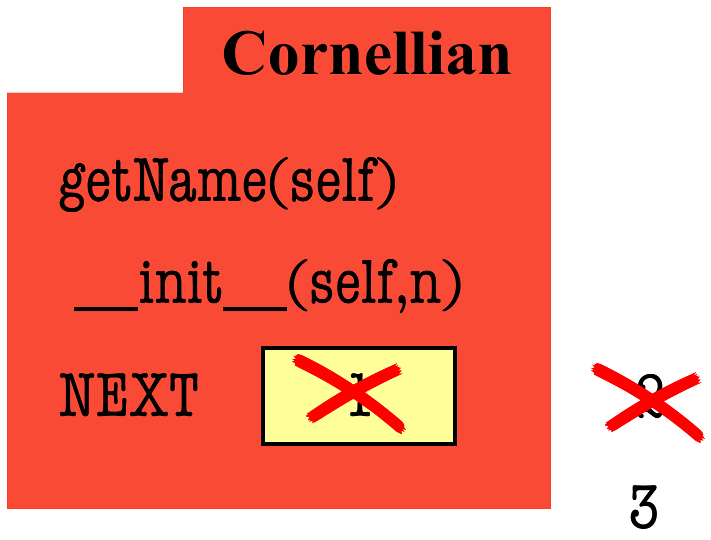


Example from Fall 2019(!)



Execute:

```
>>> a = Cornellian('Alice')
>>> b = Cornellian('Bob')
```


Cornellian
getName(self)
__init__(self,n)
NEXT
~~1~~
~~2~~
3