Review 1

Call Frames;
Diagramming Objects
The Big Issue

• Cannot answer questions on this topic unless you
  ▪ draw variables
  ▪ draw frames for function calls
  ▪ draw 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 5 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

The Frame (box) for a Function Call:

- **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

```python
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

```python
class Score(object):
    ...
    def addScore(self,value):
        """Add value to score attr"""
        self._score = self._score+value
```
def last_name_first(s):
    """Precondition: s in the form <first-name> <last-name>"""
    first = first_name(s)
    last = last_name(s)
    return last + '.' + first

def last_name(s):
    """Prec: see last_name_first"""
    end = s.find(' ')
    return s[end+1:]
def reverse(b):
    """Reverse elements of b in place (does not make a copy)
    Pre: b is a list""
    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""
    if h >= k:
        return
    temp = b[h]
    b[h] = b[k]
    b[k] = temp
    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!

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!

a

id2

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

```python
def reverse(b):
    """Reverse elements of b in place (does not make a copy)
    Pre: b is a list""
    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""
    if h >= k:
        return
temp = b[h]
b[h] = b[k]
b[k] = temp
reverse_part(b,h+1,k-1)
```

```
  a   id3
      |
   id3

  a[0]  5  3
  a[1]    7
  a[2]  3  5
```

```
  reverse
  |
  b   id3

  reverse_part
  |
  b   id3
    h  0

  reverse_part
  |
  b   id3
    h  1
    k  1

  reverse_part
  |
  b   id3
    h  2
    k  2
temp  5
```
Diagramming Objects (Folders)

Object Folder

- Folder Name (make it up)
- id4
- Instance Attributes
  - Draw attributes as named box w/ value

Class Folder

- No folder name
- classname
- Class Attributes
- Method Names

Parameters are optional in methods
```python
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`
```
3 steps to evaluating the call \( C(\text{args}) \)

- **Create a new folder** (object) of class \( C \)
  - Give it with a unique name (any number will do)
  - Folder goes into heap space
- Execute the **method** \( \text{__init__}(\text{args}) \)
- Yield the **name** 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
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
```python
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
```
a = 3

x = C(a)  # C a class

y = C(a)

x = y

class C(object):
    f = 0
    def __init__(self, k):
        self.f = k
Code Segment (with Constructors)

```python
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
```

```
  id6
  C
  f 3

  id7
  C
  f 3
```
Code Segment (with Constructors)

```python
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
```

- aliasing
Execute the call: `session()`

```python
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()))
```

```python
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
```
Execute the call: `session()`

```python
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()))
```
Execute the call: `session()`

```python
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()))
```

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 2013

class Cornellian(object):
    """Instance attributes:
    _cuid: Cornell id [int > 0]
    _name: full name [nonempty str]""
    NEXT = 1 # Class Attribute
    ...
    def _assignCUID(self):
        """Assigns _cuid to next Cornell id""
        self._cuid = Cornellian.NEXT
        Cornellian.NEXT = Cornellian.NEXT+1
    def __init__(self, n):
        """Initializer: Cornellian with name n.""
        self._name = n
        self._assignCUID()
    ...

Execute:

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

Pay close attention to class attribute NEXT
class Cornellian(object):
    """Instance attributes:
    _cuid: Cornell id [int > 0]
    _name: full name [nonempty str]"""

    NEXT = 1 # Class Attribute

    def _assignCUID(self):
        """Assigns _cuid to next Cornell id""
        self._cuid = Cornellian.NEXT
        Cornellian.NEXT = Cornellian.NEXT + 1

    def __init__(self, n):
        """Initializer: Cornellian with name n."""
        self._name = n
        self._assignCUID()

    ...
Example from Fall 2013

Execute:

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

```python
Cornellian
__init__(n)
```

`NEXT`
Example from Fall 2013

Execute:

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

```
Cornellian
__init__(n)
NEXT
```

<table>
<thead>
<tr>
<th>id10</th>
<th></th>
<th>id11</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cornellian</td>
<td></td>
<td>Cornellian</td>
</tr>
<tr>
<td>_cuid</td>
<td>1</td>
<td>_cuid</td>
</tr>
<tr>
<td>_name</td>
<td>'Alice'</td>
<td>_name</td>
</tr>
</tbody>
</table>

✗ 2

✗ 3