Exam-Prep Section 5

OOP evaluation, OOP authoring, magic methods, growth, linked lists
Recap: magic methods

def str(x):
    if x has __str__:
        return x.__str__()
    elif x has __repr__:
        return x.__repr__()
    else:
        return "<Foo object…>

def repr(x):
    if x has __repr__:
        return x.__repr__()
    else:
        return "<Foo object…>

def interpreter(x):
    print(repr(x))

def print(x):
    display str(x)

str("x") -> "x"
repr("x") -> ""x"
interpreter(str("x"))
    -> interpreter("x")
    -> print(repr("x"))
    -> print(""x"")
    -> displays "x"
Evaluation rule: call expr where optr is class

\[ x = \text{Foo}() \]

1. Look up Foo.
2. Create an empty (frame-like) box to represent the object and label it Foo object [p=class Foo].
3. Look up Foo.__init__. Call the method you find, passing the newly created object as the first argument.
4. “Return” the newly created object.
Evaluation rule: dot expr

\[ a.b \]

1. Look up \( a \). Go to the (frame-like) box that is \( a \) and look up the name \( b \).
2. If the value you end up finding is a function and \( a \) was an object, pre-bind \( a \) as the value of the function’s first argument.
3. The resulting value is the value of the dot expression.
class Foo(Bar):
    a = 1
    def f(self):
        ...

1. Look up Bar.
2. Create an empty (frame-like) box to represent the class and label it class Foo $[p=\text{class Bar}]$. 
3. Bind Foo in the current frame to the box.
4. In that frame, execute the assignment statement $a = 1$.
5. In that frame, execute the def statement $\text{def } f(\text{self})$. 
Fa14 Midterm 2 #1
OOP evaluation
25:00

- Practice OOP evaluation rules.
- See the necessity of a systematic way of keeping track of OOP on the exam.
- Practice and refine a method OOP environment diagramming.

Hints
- Do an environment diagram.
- Do an environment diagram.
- Do an environment diagram.
<table>
<thead>
<tr>
<th>Expression</th>
<th>Interactive Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>5*5</td>
<td>25</td>
</tr>
<tr>
<td>1/0</td>
<td>ERROR</td>
</tr>
<tr>
<td>Worker().work()</td>
<td>'Sir, I work'</td>
</tr>
<tr>
<td>jack</td>
<td>Peon</td>
</tr>
<tr>
<td>jack.work()</td>
<td>'Maam, I work'</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Expression</th>
<th>Interactive Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>john.work()[10:]</td>
<td>Peon, I work 'to gather wealth'</td>
</tr>
<tr>
<td>Proletariat().work(john)</td>
<td>Comrade Peon, I work Peon</td>
</tr>
<tr>
<td>john.elf.work(john)</td>
<td>'Comrade Peon, I work'</td>
</tr>
</tbody>
</table>
Break
class BinaryTree:
    
    def __init__(self, label, left, right):
        self.left = left
        self.right = right

    ● Tree
        ○ t.label is *
        ○ t.branches is list[Tree]

    ● BinaryTree
        ○ t.label is *
        ○ t.left is Tree or BinaryTree.empty
        ○ t.right is Tree or BinaryTree.empty
Terminology mismatch: label vs. entry

class Tree

    def __init__(self, label... 

• In some semesters, Trees have *label*
• In some semesters, Trees have *entry*
• They’re the same

class Tree

    def __init__(self, entry...
Fa14 Midterm 2 #4a
OOP authoring
10:00

- Practice OOP authoring.
- Learn how to actually use objects to store and manipulate information.

Hints

- Call through to parent `__init__`.
- When are you going to record a GrootTree’s parent? When it’s constructed? That’s provably impossible.
- How about when its parent is constructed?
class GrootTree(BinaryTree):
    """A binary tree with a parent."""
    def __init__(self, entry, left=BinaryTree.empty, right=BinaryTree.empty):
        BinaryTree.__init__(self, entry, left, right)
        self.parent = BinaryTree.Empty

    for b in [left, right]:
        if b is not BinaryTree.empty:
            b.parent = self
Heuristic: recurse on the branches
Fa14 Midterm 2
#4b
Recursion on trees
15:00

- Practice recursion on trees.
- Practice use of data structures to extract and store a solution.

Hints

- In a GrootTree with label \( g.label \), left branch \( g.left \), and right branch \( g.right \), when else is there no root-to-anywhere path with the entries \([x, ...]\)?
- Given that the first if-condition failed, what else do we need to know in order to assume there is exactly one such path?
- For sure, recurse on the branches in the list comprehension. But what to do with the results?
if g is BinaryTree.empty or s == [] or g.entry != s[0]:
    return 0

elif len(s) == 1 and g.entry == s[0]:
    return 1

else:
    extensions = [g.left, g.right, g.parent]

    return sum(paths(x, s[1:]) for x in extensions)
g(n)
Growth
1:00

- See that orders of growth are about understanding a function, not matching it to a pattern.

Hints
- Try it out on some representative inputs.
Θ(1)
explode(n)
Growth
2:00

- Practice composition of orders of growth.

Hints
- How long does each call to explode(n) take?
- How many iterations does the loop go through?
$\Theta(n^2)$
Growth 3:00

- Practice basic orders of growth patterns.

Hints

- Columns of 0 tiles, then 1 tile, then 2 tiles, then 3 tiles, then 4 tiles next to one another fill in half of a 4x4 square.
Θ(mn), Θ(m + n), Θ(m^2)
append, extend
Growth
3:00

- Practice orders of growth in the context of an actual problem.

Hints
- How many steps does it take to append something?
- How long does append(<list of length x>) take? What list lengths does append get called on?
$\Theta(n), \Theta(n^2)$
Heuristic: draw before and after
Sp15 Midterm 2 #3d
Linked lists
15:00

- Practice linked list manipulation.

Hints

- Mutative
- If there’s only one element left, what do you need to do?
  - Do we assume there is a duplicate before it? Or are we responsible for adding the duplicate?
- Why would we not make any changes?
- If the other if-conditions failed, what is the situation and what do we need to do?
if s is Link.empty:
    return 0

elif s.rest is Link.empty:
    s.rest = Link(s.first)
    return 1

elif s.first == s.rest.first:
    return double_up(s.rest.rest)

else:
    s.rest = Link(s.first, s.rest)
    return 1 + double_up(s.rest.rest)
Do a practice test now!!!!

Attendance:
links.cs61a.org/512