CS 61A Midterm 2 Review

Dan Wang, Chris Giola, and Jon Kotker

Eta Kappa Nu, Mu Chapter
University of California, Berkeley

3 March 2012

Scoping

What is printed after the code is executed in Python 3?

```python
x = 3
def f():
    x = 4
print(x)
```

1. 3
2. 4
3. x
4. Error

Scoping

What is printed after the code is executed in Python 3?

```
x = 3
def f():
    x = 4
print(x)
```

1. 3
2. 4
3. x
4. Error

Scoping

What is printed after the code is executed in Python 3?

```python
x = 3
def f():
    x = x + 1
print(x)
```

1. 3
2. 4
3. x
4. Error

Scoping

What is printed after the code is executed in Python 3?

```python
x = 3
def f():
    x = x + 1
print(x)
```

1. 3
2. 4
3. x
4. Error

Scoping

What is printed after the code is executed in Python 3?

```python
x = 3
def f():
    global x
    x = 4
print(x)
```

1. 3
2. 4
3. x
4. Error

Scoping

What is printed after the code is executed in Python 3?

```
x = 3
def f():
    global x
    x = 4
print(x)
```

1. 3
2. 4
3. x
4. Error

Scoping

What is printed after the code is executed in Python 3?

```
def f():
    x = 3
    def g():
        x = 4
    g()
    print(x)
f()
```

1. 3
2. 4
3. x
4. Error

Scoping

What is printed after the code is executed in Python 3?

```python
def f():
    x = 3
    def g():
        x = 4
    g()
    print(x)
f()
```

1. 3
2. 4
3. x
4. Error

Scoping

What is printed after the code is executed in Python 3?

```python
def f():
    nonlocal x
    x = 3
    def g():
        x = 4
    g()
    print(x)
f()
```

1. 3
2. 4
3. x
4. Error

Scoping

What is printed after the code is executed in Python 3?

```
def f():
    nonlocal x
    x = 3
    def g():
        x = 4
    g()
    print(x)
f()
```

1. 3
2. 4
3. x
4. Error

Scoping

What is printed after the code is executed in Python 3?

```
def f():
    x = 3
    def g():
        nonlocal x
        x = 4
    g()
    print(x)
f()
```

1. 3
2. 4
3. x
4. Error

Scoping

What is printed after the code is executed in Python 3?

```python
def f():
    x = 3
    def g():
        nonlocal x
        x = 4
    g()
    print(x)
f()
```

1. 3
2. 4
3. x
4. Error

Mutable Types

What is printed after the code is executed in Python 3?

```
x = [1, 2]
y = x
y[0] = 3
print(x[0])
```

1. 1
2. 2
3. 3
4. Error

Mutable Types

What is printed after the code is executed in Python 3?

```
x = [1, 2]
y = x
y[0] = 3
print(x[0])
```

1. 1
2. 2
3. 3
4. Error

Mutable Types

What is printed after the code is executed in Python 3?

```
x = [1, 2]
y = [x, 3]
y[0] = [4, 5]
print(x)
```

1. `[4, 5]`
2. `[1, 2]`
3. `[[4, 5], 2]`
4. Error

Mutable Types

What is printed after the code is executed in Python 3?

```
x = [1, 2]
y = [x, 3]
y[0] = [4, 5]
print(x)
```

1. [4, 5]
2. [1, 2]
3. [[4, 5], 2]
4. Error

Mutable Types

What is printed after the code is executed in Python 3?

```
x = [1, 2]
y = [x, 3]
y[0][0] = [4, 5]
print(x)
```

1. [4, 5]
2. [1, 2]
3. [[4, 5], 2]
4. Error

Mutable Types

What is printed after the code is executed in Python 3?

```
x = [1, 2]
y = [x, 3]
y[0][0] = [4, 5]
print(x)
```

1. [4, 5]
2. [1, 2]
3. [[4, 5], 2]
4. Error

Recursion

What is printed after the code is executed in Python 3?

```python
def foo1(a,b):
    if b == 0:
        return 0
    else:
        return a + foo1(a,b-1)

print(foo1(4,3))
```

1. 4
2. 7
3. 12
4. Error

Recursion

What is printed after the code is executed in Python 3?

```python
def foo1(a,b):
    if b == 0:
        return 0
    else:
        return a + foo1(a,b-1)

print(foo1(4,3))
```

1. 4
2. 7
3. 12
4. Error

Recursion

What is printed after the code is executed in Python 3?

```
def foo2(a,b):
    if b == 0:
        return 0
    else:
        return a + foo2(a,b-2)

print(foo2(4,3))
```

1. 4
2. 7
3. 12
4. Error

Recursion

What is printed after the code is executed in Python 3?

```
def foo2(a,b):
    if b == 0:
        return 0
    else:
        return a + foo2(a,b-2)

print(foo2(4,3))
```

1. 4
2. 7
3. 12
4. Error

Recursion

What is printed after the code is executed in Python 3?

```
def foo3(a,b):
    if b == 0:
        return 0
    else:
        return a + foo3(a,b)

print(foo3(4,3))
```

1. 4
2. 7
3. 12
4. Error

Recursion

What is printed after the code is executed in Python 3?

```
def foo3(a,b):
    if b == 0:
        return 0
    else:
        return a + foo3(a,b)

print(foo3(4,3))
```

1. 4
2. 7
3. 12
4. Error

Designing Recursive Algorithms

Design a recursive algorithm that computes the number of ways to choose k objects out of n objects.

```
def nChooseK(n, k):
```

Designing Recursive Algorithms

Solution:

```
def nChooseK(n, k):
    if k == 0:
        return 1
    elif k == n:
        return n
    else:
        return nChooseK(n-1,k) + nChooseK(n-1,k-1)
```

Designing Recursive Algorithms

nChooseK.png

Multiple Recursion

This problem is modified from lab 8, Exercise 4.
Consider an insect in an M by N grid. The insect starts at the left corner, $(0, 0)$, and wants to end up at the bottom right corner, $(M - 1, N - 1)$. Every minute, the insect is capable of moving right or down *any number of squares between* 1 *and* l. Write a function that determines the number of ways the insect can go from start to finish if we regard each minute as a distinct state.

```
def countpaths(M,N, l):
```

insect.jpeg

Multiple Recursion

Below is an example of calling countpaths(3,2,2)

`countpaths.png`

Multiple Recursion

```
def countpaths(M,N,l):
    return helper(M,N,l,l,0,0)

def helper(M,N,l,steps,x,y):
    if x == M - 1 and y == N - 1:
        return 1
    elif x > M-1 or y > M - 1:
        return 0
    else:
        if steps == 0:
            return 0
        else:
            return helper(M,N,l,steps-1,x,y) +
                helper(M,N,l,l,x+steps,y) +
                helper(M,N,l,l,x,y+steps)
```

Order of Growth

What is the order of growth for the following function?

```
def fib1(n):
    if n == 1:
        return 1
    elif n == 2:
        return 1
    else:
        return fib(n-1)+fib(n-2)
```

1. $O(1)$
2. $O(\log n)$
3. $O(n)$
4. $O(n^2)$
5. $O(2^n)$

Order of Growth

What is the order of growth for the following function?

```
def fib1(n):
    if n == 1:
        return 1
    elif n == 2:
        return 1
    else:
        return fib(n-1)+fib(n-2)
```

1. $O(1)$
2. $O(\log n)$
3. $O(n)$
4. $O(n^2)$
5. $O(2^n)$

Order of Growth

What is the order of growth for the following function?

```
def fib2(n):
    ls = [1,1]
    for i in range(2,n):
        ls.append(ls[-1]+ls[-2])
    return ls[-1]
```

1. $O(1)$
2. $O(\log n)$
3. $O(n)$
4. $O(n^2)$
5. $O(2^n)$

Order of Growth

What is the order of growth for the following function?

```
def fib2(n):
    ls = [1,1]
    for i in range(2,n):
        ls.append(ls[-1]+ls[-2])
    return ls[-1]
```

1. $O(1)$
2. $O(\log n)$
3. $O(n)$
4. $O(n^2)$
5. $O(2^n)$

Order of Growth

What is the order of growth for the following function?

```python
def foo(n):
    if n <= 1:
        return 0
    else:
        return 1 + foo(n/2)
```

1. $O(1)$
2. $O(\log n)$
3. $O(n)$
4. $O(n^2)$
5. $O(2^n)$

Order of Growth

What is the order of growth for the following function?

```
def foo(n):
    if n <= 1:
        return 0
    else:
        return 1 + foo(n/2)
```

1. $O(1)$
2. $O(\log n)$
3. $O(n)$
4. $O(n^2)$
5. $O(2^n)$

Classes

Convert the following below-the-line implementation of a class representing a point on the cartesian plane to a Python 3 class:

```
from math import *
def make_point(x, y):
    def point(op, *opnds):
        nonlocal x, y
        if op == 'distance_from_origin' and len(opnds) == 0:
            return sqrt(pow(x, 2) + pow(y, 2))
        elif op = 'distance_from_point' and len(opnds) == 1:
            return sqrt(pow(x - opnds[0]('x'), 2)
                + pow(y - opnds[0]('y'), 2))
        elif op = 'x' and len(opnds) == 0:
            return x
        elif op = 'y' and len(opnds) == 0:
            return y
        else:
            raise ValueError()
    return point
```

Classes

Solution

```
from math import *
class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y

    def distance_from_origin(self):
        return sqrt(pow(self.x, 2) + pow(self.y, 2))

    def distance_from_point(self, p):
        return sqrt(pow(self.x-p.x, 2) + pow(self.y-p.y, 2))
```

Identifying Parts of Classes

Consider the following class:

```python
class Foo:
    x = 3
    def __init__(self, var):
        self.y = var

    def bar(self, z):
        Foo.x = Foo.x + 1
        return self.y + z

f = Foo()
```

Identify variables that reference

1. A class:
2. An instance variable:
3. A static variable:
4. A method:
5. A parameter:
6. An object:

Identifying Parts of Classes

Consider the following class:

```python
class Foo:
    x = 3
    def __init__(self, var):
        self.y = var

    def bar(self, z):
        Foo.x = Foo.x + 1
        return self.y + z

f = Foo()
```

Identify variables that reference

1. A class: Foo
2. An instance variable:
3. A static variable:
4. A method:
5. A parameter:
6. An object:

Identifying Parts of Classes

Consider the following class:

```python
class Foo:
    x = 3
    def __init__(self, var):
        self.y = var

    def bar(self, z):
        Foo.x = Foo.x + 1
        return self.y + z

f = Foo()
```

Identify variables that reference

1. A class: Foo
2. An instance variable: y
3. A static variable:
4. A method:
5. A parameter:
6. An object:

Identifying Parts of Classes

Consider the following class:

```python
class Foo:
    x = 3
    def __init__(self, var):
        self.y = var

    def bar(self, z):
        Foo.x = Foo.x + 1
        return self.y + z

f = Foo()
```

Identify variables that reference

1. A class: Foo
2. An instance variable: y
3. A static variable: x
4. A method:
5. A parameter:
6. An object:

Identifying Parts of Classes

Consider the following class:

```python
class Foo:
    x = 3
    def __init__(self, var):
        self.y = var

    def bar(self, z):
        Foo.x = Foo.x + 1
        return self.y + z

f = Foo()
```

Identify variables that reference

1. A class: Foo
2. An instance variable: y
3. A static variable: x
4. A method: bar, __init__
5. A parameter:
6. An object:

Identifying Parts of Classes

Consider the following class:

```python
class Foo:
    x = 3
    def __init__(self, var):
        self.y = var

    def bar(self, z):
        Foo.x = Foo.x + 1
        return self.y + z

f = Foo()
```

Identify variables that reference

1. A class: Foo
2. An instance variable: y
3. A static variable: x
4. A method: bar, __init__
5. A parameter: self, var, z
6. An object:

Identifying Parts of Classes

Consider the following class:

```python
class Foo:
    x = 3
    def __init__(self, var):
        self.y = var

    def bar(self, z):
        Foo.x = Foo.x + 1
        return self.y + z

f = Foo()
```

Identify variables that reference

1. A class: Foo
2. An instance variable: y
3. A static variable: x
4. A method: bar, __init__
5. A parameter: self, var, z
6. An object: f

Classes

It is 2001 and you are a college student at Cal. You decide to create FACEPALM, an application for the Palm Pilot that maintains information about different people in your address book. FACEPALM will have a *profile* for each person. You decide to write a class called `Profile` that simulates a FACEPALM profile. It stores a person's name, the person's institution, and a list of profiles of the person's friends. It also has the following methods:

1. `add_friend(profile)` adds the given profile to the list of profiles of friends, if the profile is not already present.
2. `get_name()` returns the person's name.
3. `get_inst()` returns the person's institution.

Write the definition of the `Profile` class as specified above. We have given you a template.

```
class Profile:
    def __init__(self, name, inst):
        pass
```

Classes

Solution:

```python
class Profile:
    def __init__(self, name, inst):
        self.name = name
        self.inst = inst
        self.friends = []

    def add_friend(self, profile):
        if profile not in self.friends:
            self.friends.append(profile)

    def get_name(self):
        return self.name

    def get_inst(self):
        return self.inst
```

Dictionaries

You now want to give a user a sense of the geographical distribution of their friends. To achieve that, you provide a method `friends_in_inst`, which returns a dictionary that maps an institution to a list of the profiles of all the friends at that institution. Write the definition of `friends_in_inst`.

```
class Profile:
    ...
    def friends_in_inst(self):
        ''' FILL ME IN '''
        pass
```

Dictionaries

Solution:

```
class Profile:
    ...
    def friends_in_inst(self):
        result = {}
        for friend_profile in self.friends:
            inst = friend_profile.inst
            if inst in result:
                result[inst].append(friend_profile)
            else:
                result[inst] = [friend_profile]
        return result
```

Inheritance

You are not earning too much money from the application. In an attempt to get some revenue, you decide that profiles will be "restricted" by default: when restricted, users are only allowed to add 100 friends, beyond which they will not be allowed to add more friends. You then offer "paid" profiles that lift this restriction.

1. Modify the definition of `add_friend` in the class `Profile` that implements the restriction.
2. Define another class called `Paid_profile` that mimics the `Profile` class, except in the behavior of the `add_friend` method. (*Hint*: You should not have to rewrite a lot of `Profile`.)

Inheritance

Solution:

```
class Profile:
    ...
    def add_friend(self, profile):
        if profile not in self.friends:
            if len(self.friends) < 100:
                self.friends.append(profile)
            else:
                print "Cannot add more than 100
                       friends. Please upgrade."

class Paid_profile(Profile):
    def add_friend(self, profile):
        if profile not in self.friends:
            self.friends.append(profile)
```

Epilogue

The application, unfortunately, dies out because you have not had a chance to move it to other platforms. However, in 2003, a Harvard student named Mark Zuckerberg comes up with a suspiciously similar idea...

Destructive map

Write a destructive method d_map() that takes in a function f and a list l and changes the list so that each element e is changed to f(e). For example,

```
>>> l = [1, 2, 3]
>>> d_map(lambda x: x + 1, [1, 2, 3])
>>> l
[2, 3, 4]
```

Destructive map

Solution

```python
def d_map(f, l):
    for i in range(len(l)):
        l[i] = f(l[i])
```

Memoization

Consider the mapping of the numbers $1, 2, \ldots, 26$ to the letters where 1 maps to `A`, 2 maps to `B`, and so on.
Given a string of numbers, how many ways are there to insert spaces such that all the numbers correspond to valid letters (i.e., are in $\{1, 2, \ldots, 26\}$)? For example, for the string `'1012'`, there are two ways:

- 10, 1, 2
- 10, 12

The splitting into `1, 0, 12` is not valid because `0` does not correspond to a letter. Also, the splitting into `1, 01, 2` is not valid because `01` does not correspond to a letter.

Memoization

The following function definition is a recursive solution. This function is very inefficient. Write a version that uses memoization to reduce the number of recursive calls.

```python
def num_of_splits(s):
    if len(s) == 0:
        return 1
    else:
        return (check1(s) * num_of_splits(s[1:]))
            + (check2(s) * num_of_splits(s[2:]))

def check1(s):
    return s[0] in '123456789'

def check2(s):
    if len(s) > 1:
        if s[0] == '1':
            return s[1] in '0123456789'
        elif s[0] == '2':
            return s[1] in '0123456'
    return False
```

Memoization

Solution:

```python
def num_of_splits2(s):
    m = {}
    def helper(s):
        if s in m:
            return m[s]
        elif len(s) == 0:
            return 1
        else:
            m[s] = (check1(s) * helper(s[1:])) +
                (check2(s) * helper(s[2:]))
            return m[s]
    return helper(s)
```