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?

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

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

Scoping

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

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

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

Scoping

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

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

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

Scoping

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

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

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

Scoping

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

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

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

Scoping

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

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

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

Scoping

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

```
1 def f():
2     x = 3
3     def g():
4         x = 4
5     g()
6     print(x)
7 f()
```

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

Scoping

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

```
1 def f():
2     x = 3
3     def g():
4         x = 4
5     g()
6     print(x)
7 f()
```

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

Scoping

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

```
1 def f():
2     nonlocal x
3     x = 3
4     def g():
5         x = 4
6     g()
7     print(x)
8 f()
```

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

Scoping

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

```
1 def f():
2     nonlocal x
3     x = 3
4     def g():
5         x = 4
6     g()
7     print(x)
8 f()
```

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

Scoping

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

```
1 def f():
2     x = 3
3     def g():
4         nonlocal x
5         x = 4
6     g()
7     print(x)
8 f()
```

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

Scoping

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

```
1 def f():
2     x = 3
3     def g():
4         nonlocal x
5         x = 4
6     g()
7     print(x)
8 f()
```

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

Mutable Types

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

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

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

Mutable Types

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

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

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

Mutable Types

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

```
1 x = [1, 2]
2 y = [x, 3]
3 y[0] = [4, 5]
4 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?

```
1 x = [1, 2]
2 y = [x, 3]
3 y[0] = [4, 5]
4 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?

```
1 x = [1, 2]
2 y = [x, 3]
3 y[0][0] = [4, 5]
4 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?

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

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

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

```
1  from math import *
2  class Point:
3      def __init__(self, x, y):
4          self.x, self.y = x, y
5
6      def distance_from_origin(self):
7          return sqrt(pow(self.x, 2) + pow(self.y, 2))
8
9      def distance_from_point(self, p):
10         return sqrt(pow(self.x-p.x, 2) + pow(self.y-p.y, 2))
```

Identifying Parts of Classes

Consider the following class:

```
1  class Foo:
2      x = 3
3      def __init__(self, var):
4          self.y = var
5
6      def bar(self, z):
7          Foo.x = Foo.x + 1
8          return self.y + z
9
10 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:

```
1  class Foo:
2      x = 3
3      def __init__(self, var):
4          self.y = var
5
6      def bar(self, z):
7          Foo.x = Foo.x + 1
8          return self.y + z
9
10 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:

```
1  class Foo:
2      x = 3
3      def __init__(self, var):
4          self.y = var
5
6      def bar(self, z):
7          Foo.x = Foo.x + 1
8          return self.y + z
9
10 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:

```
1  class Foo:
2      x = 3
3      def __init__(self, var):
4          self.y = var
5
6      def bar(self, z):
7          Foo.x = Foo.x + 1
8          return self.y + z
9
10 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:

```
1  class Foo:
2      x = 3
3      def __init__(self, var):
4          self.y = var
5
6      def bar(self, z):
7          Foo.x = Foo.x + 1
8          return self.y + z
9
10 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:

```
1  class Foo:
2      x = 3
3      def __init__(self, var):
4          self.y = var
5
6      def bar(self, z):
7          Foo.x = Foo.x + 1
8          return self.y + z
9
10 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:

```
1  class Foo:
2      x = 3
3      def __init__(self, var):
4          self.y = var
5
6      def bar(self, z):
7          Foo.x = Foo.x + 1
8          return self.y + z
9
10 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

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

```
1 def d_map(f, l):
2     for i in range(len(l)):
3         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.

```
1  def num_of_splits(s):
2      if len(s) == 0:
3          return 1
4      else:
5          return (check1(s) * num_of_splits(s[1:]))
6              + (check2(s) * num_of_splits(s[2:]))
7
8  def check1(s):
9      return s[0] in '123456789'
10
11 def check2(s):
12     if len(s) > 1:
13         if s[0] == '1':
14             return s[1] in '0123456789'
15         elif s[0] == '2':
16             return s[1] in '0123456'
17     return false
```