Quiz #3 (Python)

CS671

due 25 July 2013 6:10pm

1. Data Structures:

   Write a function `pyMap` that emulates SML's `map`, but does so using only list comprehensions and can be used on any iterable data structure (list, tuple, string, etc.)

```
def pyMap(f,l):
   toRet = []
   for i in l:
      toRet.append(f(i))
   return toRet
```

2. Variable Scoping:

   Write a function `printVal` that takes a string `val` as an argument, allowing it to default to `None`. It then looks for a variable named `val` with a value (that is not `None`) in the following order of scopes: global, function namespace, parameters/local

```
def printVal(val=None):
   if val == None:
      return val
   elif val in globals() and globals()[val] != None:
      return globals()[val]
   elif val in locals() and locals()[val] != None:
      return locals()[val]
   else:
      return None
```

3. Classes & Objects:

   Write a class `Point2D` with `x` and `y` values and another class `Point3D` with `x`, `y`, and `z` values. Give each class a constructor that takes these values.

```
class Point2D(object):
   def __init__(self,x,y):
      self.x = x
      self.y = y
```

```
class Point3D(object):
  def __init__(self,x,y,z):
    self.x = x
    self.y = y
    self.z = z
```

```
p2d = Point2D(3, 3)
p3d = Point3D(1, 1, 1)
p3d - p3d # returns a Point3D with coords (0, 0, 0)
p2d - p3d # returns a Point3D with coords (2, 2, -1)
p3d - p2d # returns a Point3D with coords (-2, -2, 1)
p3d - 10 # errors
```

Write the code that would need to be added to `Point2D` and/or `Point3D` to make the above code run or error as described.

```
class Point2D(object):
   def __init__(self,x,y):
      self.x = x
      self.y = y
   def __sub__(self,other):
      if isinstance(other,Point2D):
         return Point2D((self.x - other.x),
                        (self.y - other.y))
      elif isinstance(other,Point3D):
         return Point3D((self.x - other.x),
                       (self.y - other.y),
                       (0 - other.z))
      else:
         raise TypeError("{} is an invalid type".format(type(other)))
```

```
class Point3D(object):
   def __init__(self,x,y,z):
      self.x = x
      self.y = y
      self.z = z

   def __sub__(self,other):
       if isinstance(other,Point2D):
          return Point3D((self.x - other.x),
                         (self.y - other.y),
                         (self.z - 0))
      elif isinstance(other,Point3D):
          return Point3D((self.x - other.x),
                          (self.y - other.y),
                          (self.z - other.z))
      else:
         raise TypeError("{} is an invalid type".format(type(other)))
```

4. Metaprogramming:

```
class A:
  def __init__(self, name):
    self.name = name
  def __repr__(self):
    return self.name
a = A("a’s name")
b = A("b’s name")
```

Use metaprogramming to make it so a’s string representation is in all uppercase, while `b`’s (and all other `A` objects’) are in all lowercase.

```
a = A("a's name")
b = A("b's name")
a.__dict__['name'] = a.__dict__['name'].upper()
```

Write a `@keepSum` decorator that will print a running sum of the values returned by a function after each call.

```
def keepSum(fn):
    def doFunc(*args):
        if 'total' not in doFunc.__dict__:
            doFunc.__dict__['total'] = fn(*args)
        else:
            doFunc.__dict__['total'] += fn(*args)
        return doFunc.__dict__['total']
    return doFunc
```

```
def series(f, n, end):
  if n < end:
    return f(n) + series(f, n+1, end)
  else:
    return 0
```

What will happen to this function if we use the `@keepSum` decorator on it? Rewrite it so the decorator will behave more as we would usually prefer.

```
def keepSum(fn):
    def doFunc(*args):
        if 'total' not in doFunc.__dict__:
            doFunc.__dict__['total'] = fn(*args)
        else:
            doFunc.__dict__['total'] += fn(*args)
        return doFunc.__dict__['total']
    return doFunc
```

5. Lazy programming:

   Write a function `randConverge(mn, mx)` that returns a generator object for integers in the range between `mn` and `mx` which converges to `mn`. The integer generated/returned from the previous step should become the `mx` for the next, until `mn` and `mx` are the same. So, the last integer from the generator should always be `mn`, and `mn` should never be generated twice.

```
def randConverge(mn,mx):
    while mx > mn:
        toRet = random.randrange(mn,mx)
        yield toRet
        mx = toRet
    yield None
```

   Write a function `firstFive(g)` that returns a list containing the first five items from generator `g`, placing `None` in any indices where the generator has run out.

```
def firstFive(g):
    import itertools
    return list(itertools.islice(g,5))
```