CS 61A Structure and Interpretation of Computer Programs
Fall 2016

MIDTERM 2

1. (2 points) Route Cipher

Fill in the `encrypt` and `decrypt` methods of the `RouteCipher` class, making sure to utilize inheritance whenever possible. The functionality of a route cipher is defined in its docstring. Note that these methods must both record their argument history and assert that their inputs are strings.

Unfortunately, `RouteCipher` happens to be ridiculously difficult (well, for a quiz question at least) and generally requires tricky edge case coverage for total robustness. Furthermore, if anyone manages to fit a correct implementation into the space I allotted, then they are probably not actually of this Earth. Therefore, feel free to ignore `RouteCipher` and just do the "true/false" question on the back instead.

```
class Cipher:

    def __init__(self):
        self.plainhist = {} # {plaintext: # times encrypted}
        self.cipherhist = {} # {ciphertext: # times "decrypted"}

    def encrypt(self, plaintext):
        assert type(plaintext) == str, 'input must be a string'
        self.plainhist[plaintext] = self.plainhist.get(plaintext, 0) + 1
        if self.plainhist.get(plaintext, 0) > 10:
            print("...you've encrypted this %d times"
                  % self.plainhist[plaintext])
        return plaintext

    def decrypt(self, ciphertext):
        assert type(ciphertext) == str, 'input must be a string'
        self.cipherhist[ciphertext] = self.cipherhist.get(ciphertext, 0) + 1
        if self.cipherhist.get(ciphertext, 0) > 10:
            print('dude get a life')
        return ciphertext

class RouteCipher(Cipher):
    """Standard route cipher. For encryption, writes plaintext out
    as characters in a rectangular grid, then reads off elements
    a spiraling inward, clockwise fashion (starting at the top left).

    For example, HELLO WORLD would be displayed in a two-row grid as

    H L O W R D
    E L   O L _

    and would be encrypted as HLOWRD_LO LE.
    """
    def __init__(self, num_rows):
        assert num_rows > 0, 'row count must be positive'
        Cipher.__init__(self)
        self.num_rows = num_rows
```

```
def encrypt(self, plaintext):
# BEGIN SOLUTION -->









# <-- END SOLUTION

def decrypt(self, ciphertext):
# BEGIN SOLUTION -->









# <-- END SOLUTION
```

2. (8 points) True or False

Is it `True`? Or is it `False`? You decide!

(a) If you want to call a bound method, then you must explicitly pass in an argument as the `self` parameter.

(b) You can define a normal function (i.e. the kind we've been using all year) within a class, and access it without the use of dot notation.

(c) You can access instance attributes from within a *non*-bound method if you call it from the body of a bound method.

(d) All user-defined classes are technically subclasses.

(e) If a function defined in a class takes `self` as its first argument, then it must always be called using dot notation with an *instance* on the left side of the dot.

(f) `self` is a special name in Python. If you were to use, say, `myself` as a method's first parameter name, then things would break.

(g) If you change something in a subclass, then that change will propagate to all instances of the base class.

(h) In general, it's fine to replace an instance on the left side of a dot expression with `self`.