CPS 109 - Review

Agenda

1. Reading/Deciphering questions
2. Recursion
3. Classes/OOP
4. Inheritance
5. Testing
6. Guest Speaker!

Understanding Questions

It came to my attention that everyone is having a bit of trouble understanding what a problem or question is looking for.

Understanding Questions

```
def q3(x, epsilon) :
    '''
    Returns guess, the approximate square root of x, such that
    abs(guess**2 - x) < epsilon using Heron's algorithm, where
    start with guess = x / 2 and improve guess to be (guess + x / guess) / 2
    '''
    pass
```

Understanding Questions

```
class myTests(unittest.TestCase):
    def test0(self):
        self.assertTrue(abs(q3(4, 0.1) - 2) <= 0.1)
    def test1(self):
        self.assertTrue(abs(q3(99, 0.1) - math.sqrt(99)) <= 0.1)
    def test2(self):
        self.assertTrue(abs(q3(999, 0.1) - math.sqrt(999)) <= 0.1)
    def test3(self):
        self.assertTrue(abs(q3(1, 0.1) - 1) <= 0.1)
    def test4(self):
        self.assertTrue(abs(q3(0.25, 0.01) - 0.5) <= 0.01)
```

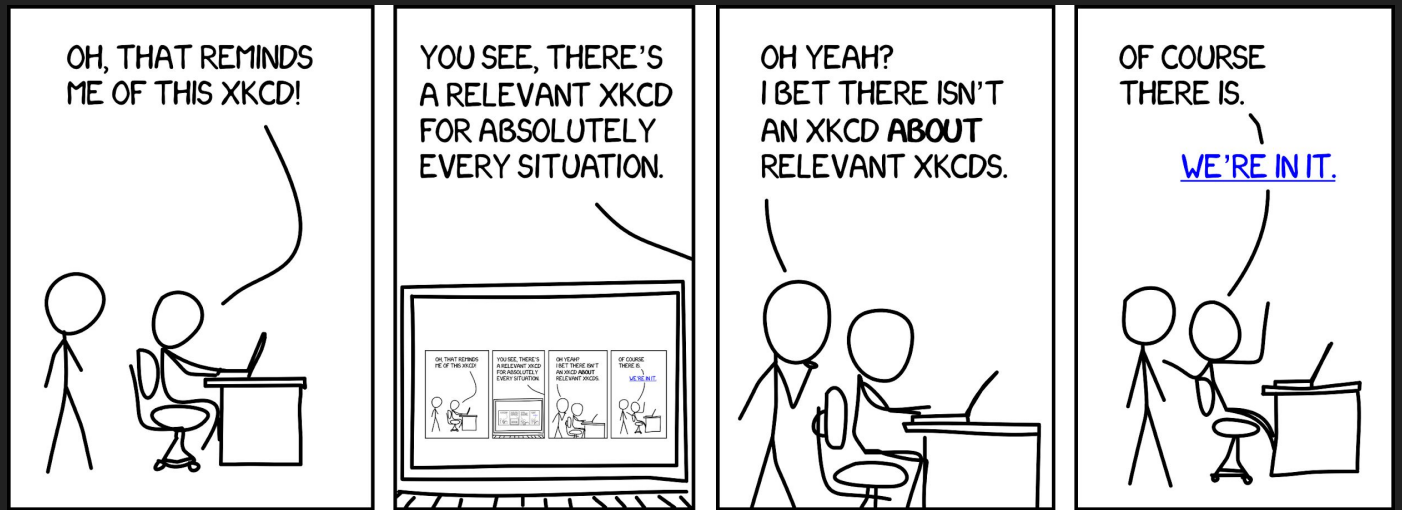
A Note On Mutability

Something is **mutable** if its contents can be changed (like a list).

Something is **immutable** if its contents cannot be changed (like a string or a tuple) and instead you have to overwrite it.

Recursion

Recursion is a special sort of “loop”, so to speak. It refers to a type of function that can call itself in its execution.



Recursion

How do you design a recursive function? Here are my suggested steps:

1. Make sure your function has a clear goal
2. Write out (then type up) your 1 or more base cases.
3. Design your recursive step.

Object Oriented Programming

Also known as OOP(there it is).

The long as short of OOP is really quite simple: every single piece of data is an object. But what on Earth is an object??

Object Oriented Programming

For all intents and purposes, there are two main kinds of objects you all need to be aware of:

1) Native objects (the datatypes you all love and know)
2) Classes you've defined yourselves

Classes

Recall that classes are custom data types that you can make yourself! They can contain any and all member variables so long as you declare the class properly.

They also contain any and all member/instance methods (internal functions).

Classes

```
class Rectangle(object) :
    '''This class represents a rectangle in the x-y coordincate system
    where the edges of the rectangle are aligned with the x- and y- axes.
    A Rectangle object has data attributes lowerleft and upperright,
    which are tuples representing the (x, y) coordinates of the lower
    left corner and the upper right corner.
    '''
    def __init__(self, x1, y1, x2, y2) :
        '''Assumes the (x1, y1) are the coordinates of the lower left corner
        and (x2, y2) are the coordinates of the upper right corner.
        '''
        self.lowerleft = (x1, y1)
        self.upperright = (x2, y2)
```

Classes Problem Walkthrough

```
'''
Story: The Teletubbies are vanquished, but in the realm of dreams they still haunt
you! Your task is to implement the following classes and helper methods according
to the requirements described in each class/method, so as to conquer your nightmares,
and obliterate all traces of the Teletubbies!
'''

class Dream:
    '''
    Dreams are pretty good. Usually with apple pies and candies and all that good stuff.

    Your task is as follows:
    Construct the "__init__" method for a Dream class, that takes in 2 arguments (plus self),
    self (which you ALWAYS require in your constructors, even if there is nothing else in them!)
    a string "title", and
    an int "length"
    And sets a list dream_elements to start as the empty list: []
    [1]

    Then, once you have completed your constructor, create an instance method called
    change_title(a), which changes the title of your dream instance to the string a.
    [2, 3]

    Then, create another instance method called add_to_dream(topic), which adds the string
    topic to the dream_elements list.
    [2, 3]

    Finally, create an instance method dreams_to_dust(), which replaces each element of the
    dream_elements list with the string "dust".
    [2, 3]
    '''
    pass
```

How to decompose a “Class” exam problem

1: Identify what variables your class has
2: Implement the “__init__” method
3: Identify what instance methods you will need to implement (pay special attention to the input arguments and expected return values)
4: Implement the instance methods

Constructor Criticism

```python
class Dream:
    '''
    Dreams are pretty good. Usually with apple pies and candies and all that good stuff.

    Your task is as follows:
    Construct the "__init__" method for a Dream class, that takes in 2 arguments (plus self),
    self (which you ALWAYS require in your constructors, even if there is nothing else in them!)
    a string "title", and
    an int "length"
    And sets a list dream_elements to start as the empty list: []
    '''
    def __init__(self, title, length):
        self.title = title
        self.length = length
        self.dream_elements = []
```

```python
d = Dream("Dream Where I Pass My Exam", 4)
```

```python
class Nightmare:
    '''
    Nightmares are pretty bad. Usually with Teletubbies and chainsaws and all that bad stuff.

    Your task is as follows:
    Construct the "__init__" method for a Nightmare class, that takes in 0 arguments (except self of course)!
    However, you should create an instance variable "topic" that defaults to the string
    "Teletubbies", and an instance variable for an attached Dream that the nightmare is
    in, called "attached_dream" (which defaults to None).
    [1]
    '''
    def __init__(self):
        self.topic = "Teletubbies"
        self.attached_dream = None
```

```python
n = Nightmare()
```

How to make the __init__:

1: Write out the first line, including self and any other input arguments
2: Set self.internal_variables to input arguments
3: Create any default internal variables
4: (Optional) Perform any required behaviour

How to handle empty constructors:

1: Make sure you include self in the brackets
2: Everything else should be in the __init__ body

Unittests as a Guide

```
27  class Dream:
28      '''
29      Dreams are pretty good. Usually with apple pies and candies and all that good stuff.
30
31      Your task is as follows:
32      Construct the "__init__" method for a Dream class, that takes in 2 arguments (plus self),
33      self (which you ALWAYS require in your constructors, even if there is nothing else in them!)
34      a string "title", and
35      an int "length"
36      And sets a list dream_elements to start as the empty list: []
37      '''
38      def __init__(self, title, length):
39          self.title = title
40          self.length = length
41          self.dream_elements = []
```

```
d = Dream("Dream Where I Pass My Exam", 4)
```

```
66  class Nightmare:
67      '''
68      Nightmares are pretty bad. Usually with Teletubbies and chainsaws and all that bad stuff.
69
70      Your task is as follows:
71      Construct the "__init__" method for a Nightmare class, that takes in 0 arguments (except self of course)!
72      However, you should create an instance variable "topic" that defaults to the string
73      "Teletubbies", and an instance variable for an attached Dream that the nightmare is
74      in, called "attached_dream" (which defaults to None).
75      [1]
76      '''
77      def __init__(self):
78          self.topic = "Teletubbies"
79          self.attached_dream = None
```

```
n = Nightmare()
```

```
141 class myTests(unittest.TestCase):
142     def test1(self): # Testing Dream constructor
143         d = Dream("my awesome dream", 5)
144         self.assertEqual(d.title, "my awesome dream")
145         self.assertEqual(d.length, 5)
146         self.assertEqual(d.dream_elements, [])
147     def test2(self): # Testing nightmare constructor
148         n = Nightmare()
149         self.assertEqual(n.topic, "Teletubbies")
150         self.assertEqual(n.attached_dream, None)
```

Function Flattery

```
    Then, once you have completed your constructor, create an instance method called
    change_title(a), which changes the title of your dream instance to the string a.
    [2, 3]

    Then, create another instance method called add_to_dream(topic), which adds the string
    topic to the dream_elements list.
    [2, 3]

    Finally, create an instance method dreams_to_dust(), which replaces each element of the
    dream_elements list with the string "dust".
    [2, 3]
    '''
    def change_title(self, a):
        self.title = a

    def add_to_dream(self, topic):
        self.dream_elements.append(topic)

    def dreams_to_dust(self):
        for element in range(len(self.dream_elements)):
            self.dream_elements[element] = "dust"
```

```
    def test3(self): # Testing Dream.change_title()
        d = Dream("my awesome dream", 5)
        d.change_title("my great dream")
        self.assertEqual(d.title, "my great dream")
    def test4(self): # Testing Dream.add_to_dream()
        d = Dream("my awesome dream", 5)
        d.change_title("my great dream")
        d.add_to_dream("rainbows")
        d.add_to_dream("butterflies")
        self.assertEqual(d.dream_elements, ["rainbows", "butterflies"])
```

How to create an instance function:

1: The first argument should always be self

2: Whenever you want to access a variable from the instance of a class, use the syntax: self.variable. To modify, use: self.variable = new_value

3: Treat it like any other function! If you need to return, then return. If you need to perform some logic, then do so! Don't get scared!

Unittests are always helpful!

Funception

```
    Then, create a method attach_to_dream(d), which attached your nightmare instance
    to a dream via setting attached_dream to the argument Dream d.
    [2, 3]

    Then, create a method has_attached_dream() which returns True if your attached_dream
    is not None, and False otherwise.
    [2, 3]

    Finally, create a method WAKE_ME_UP(), which will set the attached_dream's length to 0,
    and call it's dreams_to_dust() method.
    [2, 3]
    '''
    def attach_to_dream(self, d):
        self.attached_dream = d

    def has_attached_dream(self):
        if (self.attached_dream is not None):
            return True
        else:
            return False

    def WAKE_ME_UP(self):
        if (self.has_attached_dream()):
            self.attached_dream.length = 0
            self.attached_dream.dreams_to_dust()
```

Just like any other function, there is no limit to how many times you can call an instance function, its contents, and where you can call it.

You'll notice that in the WAKE_ME_UP() function, it takes the Dream object associated with the nightmare (if it exists (which it checks via calling self.has_attached_dream()!)) and calls that Dream object's dreams_to_dust() instance function! This is totally legit yo!

‘Helper’ Me Out!

```
107 def return_titles(list_of_dreams):
108     '''
109     Create a method that takes in a list of Dreams "list_of_dreams" as input, and returns
110     a list containing the titles of all Dreams in that list.
111     [2, 4]
112     '''
113     to_return_list = []
114     for dream in list_of_dreams:
115         to_return_list.append(dream.title)
116     return to_return_list
117 
118 def WAKE_ME_UP_INSIDE(cant_wake_up):
119     '''
120     Create a method that takes in a list of Nightmares "cant_wake_up", as input, which
121     then calls the WAKE_ME_UP() method for each Nightmare in the list.
122     [3, 4]
123     '''
124     for nightmare in cant_wake_up:
125         nightmare.WAKE_ME_UP()
126 
127 def how_long_are_my_dreams(list_of_dreams):
128     '''
129     Create a method that takes in a list of Dreams "list of dreams" as input, and returns
130     the sum of all Dream lengths in that list.
131     [2, 4]
132     '''
133     total_sum = 0
134     for dream in list_of_dreams:
135         total_sum += dream.length
136     return total_sum
```

Helper functions are just normal functions that are built to interact with Classes in some manner.

Notice that we can access a class object c’s internal variable x via the syntax:
c.x
We can even modify c.x, via the syntax:
c.x = new_val

You can also call instance methods outside of a class (**where self is implicitly being assigned to the instance of the class object!**)

[illegible]() looks like:

```
102     def WAKE_ME_UP(self):
```

but self is not provided, just being implicitly assigned to nightmare. Check out the unittests for hints!

A Note for the Class

While classes are just a portion of your exam, the reason why we went into so much detail is because the general framework for such a question is applicable to ANY programming question you have on your exam.

You should always:

1. Read what the question is asking for carefully
2. Be especially careful about the inputs arguments / desired returns (types, names, etc.)
3. Use the unittests as a fallback guide when something is not super clear

Inheritance

But why use classes? I don’t even care about predefined behaviour.

Well that’s where you’re wrong, bucko. One of most powerful features of classes and OOP is that of inheritance.

Inheritance

```
class Cat(Animal):
    '''
    Child class of Animal called Cat. Since cats aren't
    brushed by default, the member variable for is_brushed is
    set to False.
    '''
    def __init__(self, name, age, fur_colour, eye_colour):
        super().__init__(name, age)
        self.fur_colour = fur_colour
        self.eye_colour = eye_colour
        self.is_brushed = False
```

Testing

Testing is insanely important yet no one really presses in why it’s important. Now, we are gonna contextualize the testing syntax you’ve seen AND make you good at writing tests.

Testing

In this course, particularly in the midterm, you've seen the predictable syntax. You declare a test class with member methods and then call it in your __main__ (remember that if your code isn't in a function or class, it goes in the main!)

This is where it all comes together...

Testing

```
class myTests(unittest.TestCase):
    def test0(self):
        self.assertEqual(outside_of(5, 6, 2), False)
    def test1(self):
        self.assertEqual(outside_of(5, 7, 1.5), True)
    def test2(self):
        self.assertEqual(outside_of(1, 2, 1), False)
    def test3(self):
        self.assertEqual(outside_of(9, 9, 0), False)
    def test4(self):
        self.assertEqual(outside_of(9, 9, 100), False)
    def test5(self):
        self.assertEqual(outside_of(-90, 100, 0), True)
```