6.0001 Final Exam Review Part 2
Notes by Amy Zhao

Topics:

– Classes, inheritance
– Exceptions
– Functions
– Recursion
Classes: basics
(vehicle.py)

• Class definitions consist of data attributes (variables) and method attributes (functions)

class Vehicle(object):
    nextIdNum = 1  # this is a data attribute

    def __init__(self):  # this is a method attribute
        self.idNum = Vehicle.nextIdNum
        Vehicle.nextIdNum += 1

    def honk(self):  # this is a method attribute
        print "HONK!"
Classes: operations

• Classes support two kinds of operations
  – instantiation
    • used to create instances of the class

        myVehicle = Vehicle()

  – attribute references
    • uses dot notation to access attributes

        print myVehicle.idNum       # print a data attribute
        myVehicle.honk()           # invoke a method attribute
Classes: data attributes

- should never be accessed directly
- always use a getter method (method should take care of cloning if the variable is a mutable type)

```python
class Vehicle(object):
    nextIdNum = 1

    def __init__(self):
        self.idNum = nextIdNum
        Vehicle.nextIdNum += 1

    def honk(self):
        print "HONK!"

    # getter method for idNum data attribute
    def getIdNum(self):
        return self.idNum
```
Classes: method attributes

• First parameter of each method is always called **self** (by convention)
  • Why? Because dot notation implicitly passes the object preceding the dot into the method as the first parameter

• There are special method names in Python
  • **__init__** is automatically invoked following instantiation
  • **__str__** is invoked when you **print** an object
  • **__lt__(self, other)** defines what happens when you use boolean comparison on instances
  • **__getitem__** is invoked when you index into an instance using []
Classes: class and instance variables

- Data and method attributes can be associated either with a class itself or with instances of the class
- We only really talk about **class variables** and **instance variables** in this course

```python
class Vehicle(object):
    nextIdNum = 1

    def __init__(self):
        self.idNum = Vehicle.nextIdNum
        Vehicle.nextIdNum += 1

    def honk(self):
        print "HONK!"

    def getIdNum(self):
        return self.idNum

print Vehicle.nextIdNum # this prints a class variable

myVehicle = Vehicle()
print myVehicle.idNum    # this prints an instance variable
```
Classes: inheritance

- **Subclasses** inherit from a **superclass**
  - Subclass inherits all attributes (data and method) of superclass
  - Subclass can also:
    - Add new attributes
    - Override attributes of the superclass

```python
class Car(Vehicle):
    # this is a new method attribute
    def openDoor(self):
        self.isDoorOpen = True

    # we’re overriding this method attribute
    def honk(self):
        print “BEEP!”
```
Classes: inheritance (cont.)

• Python keyword `pass` used for subclass with no additional attributes

```python
class Bicycle(Vehicle):
    pass
```

• Can have multiple levels of inheritance

```python
# this inherits from Car which inherits from Vehicle
class Convertible(Car):
    def openRoof(self):
        self.isRoofOfOpen = True
```
Exceptions

• **Unhandled exceptions**
  – Cause program to terminate

• **Handled exceptions**
  – You should handle exceptions if you know what exception the code might raise
Exception handling

• Handle exceptions using try...except
  – try block will execute
  – If an error is raised in the try block, your program will jump out of the try block and look for the corresponding except block
  – If there is an except block for that type of error, that except block will execute
More about errors

• Common error types
  – IndexError e.g. accessing L[4] when L is a list with 2 elements
  – TypeError e.g. int([1,2,3])
  – ZeroDivisionError is raised when you divide by 0
  – KeyError is raised when you try to access a dictionary key that is not in the dictionary

• raise keyword
  – You can raise any error

    raise ValueError

  – Particularly useful when using errors for control flow
def findStateOfCity(c):
    cities = {'Boston': 'MA', 'Seattle': 'WA'}

    try:
        if(type(c)!=str):
            raise TypeError("I'm a type error argument!")
        state = cities[c]
        print "The city", c,"is in", state,"state"
    except KeyError:
        print c, "does not exist in cities dictionary!"
    except TypeError:
        print c, "is not a string!"
    except:
        print "Unknown error!"

findStateOfCity('Cambridge')
findStateOfCity('Seattle')
findStateOfCity(3)
Classes and exceptions example

See dice.py
Functions: invocation

• Q: What happens when you invoke (call) a function?
• A: Values of actual parameters (arguments) are assigned to the formal parameters

```python
def gcd(x,y): # x and y are formal parameters
    # calculate gcd of x and y
    a = 10
    b = 15
    gcd(a,b)   # a and b are actual parameters
```
Functions: return statements

• Ways to write return statements in Python (the last 3 are equivalent):
  - `return <value>`
  - `return None`
  - `return`
  - `<nothing>`
Functions: scoping

• What is a scope?
  – A **scope** or **namespace** is a space where variables exist
  – The scope of a variable determines where the variable can be accessed from
Functions: scoping

• Each .py file has a global scope that spans the entire file
• Each function inside the file defines a new name space or scope
Functions: scoping

• Rules of scoping
  – Variables defined inside a scope (basically, inside a `def`) can only be seen by code inside that scope
  – Variables defined inside a scope are different from variables outside the scope
Functions: scoping example #1

L = [1,2]
# L is defined in the # global scope

def myFunc():
    a = 3
    b = 5
    print L # OK

def myOtherFunc():
    a = 'hi'
    print L # OK
    print b # error
    print a # error
Functions: scoping example #2

- CAN read outside variables from inside a function’s scope

```python
a = 3  # a exists in global scope

# myFunc is defined in global scope as well
def myFunc():
    # myFunc can access a because it is defined in the same scope as myFunc
    print a  # prints 3
    b = 2

print b  # this will give an error since b was defined in myFunc’s scope and can only be seen by code within myFunc’s scope
```
Functions: scoping example #3

- CANNOT change what an outside variable points to from inside the function

```python
b = 5

def myFunc():
    b = 4  # this just creates a new
    # variable b in myFunc’s
    # scope

myFunc()
print b      # prints 5
```
Functions: scoping example #4

- CAN mutate an outside object

```python
L = [1, 2, 3]

def myFunc():
    L.append(0)

myFunc()
print L
```
def makeLighter(c):
    def isPrimary():
        if(c == "red" or c == "yellow" or c == "blue"):
            # note that c belongs to the enclosing function's scope
            return True
        else:
            return False

    if isPrimary():
        color = "light " + c #Q: are we writing to the color variable outside?
        return color
    elif(c == "black"):
        color = "gray"
        return color

color = "red"
print makeLighter(color),"is lighter than",color

color = "black"
print makeLighter(color),"is lighter than",color

color = "white"
print makeLighter(color),"is lighter than",color

#Q: Why does this say None? think about return types...
Recursion

• A recursive function always has
  – at least one base case
  – at least one recursive case

• Approaches to recursion
  – Base case?
    • look for the simplest version of the problem, which should have a simple answer
  – Recursive case?
    • how do you call the function again with a simpler input? (simpler usually means smaller or shorter)
  – Do these recursive and base cases work to solve the problem?
Recursion problem #1
(palindrome.py)

- Q: write a recursive function that checks if string s is a palindrome
- Note: there are often multiple ways to write a recursive solution! I think #1 is more intuitive, but #2 is what is given in the textbook. Both solutions work.
- A #1:
  ```python
def isPalindrome(s):
    # 3 base cases
    if len(s)==1:
        return True
    elif len(s)==2 and s[0]==s[1]:
        # we will hit this if e.g. s='aa'
        return True
    elif len(s)==2 and not s[0]==s[1]:
        # we will hit this if e.g. s='ab'
        return False
    return isPalindrome(s[1:len(s)-1]) and s[0]==s[len(s)-1]
```
Recursion problem #1

• A #2:

def isPalindrome(s):
    if len(s) <= 1:
        return True
    return isPalindrome(s[1:len(s)-1]) and s[0] == s[len(s)-1]
Recursion problem #2 (stairs.py)

Q: I want to go up a flight of stairs that has n steps. I can either take 1 or 2 steps each time. How many different ways can I go up this flight of stairs? Write a function count_stair_ways that solves this problem for me.

A:

def count_stair_ways(n):
    # 2 base cases
    if n==1:
        return 1
    elif n==2:
        return 2

    # recursive case
    return count_stair_ways(n-1) + count_stair_ways(n-2)

print count_stair_ways(3)  # 1,1,1; 1,2; 2,1
print count_stair_ways(4)  # 1,1,1,1; 1,1,2; 1,2,1; 2,1,1; 2,2