Lecture 22

The Importance of Encapsulation; Dictionaries; Python Magic; List Comprehensions

1. The Importance of Encapsulation

As we said before, most classes are designed so that, when client code *uses* the class, it will not refer directly to attritube variables of objects, but rather it will instead use methods to change/access those objects. This means that you will see less bits of code that look like

`x._attribute = value`

and more bits like

`x.set_attribute(value)`

(notice the parentheses on this line!). The reasoning is roughly that "the attribute variables are the technical underside, while the methods are the usable interface". Indeed, that's what those underscores are about in front of the attribute names: they are a convention, a gentle reminder that's a bad idea to tinker with the details, and to stick to the user-friendly interface instead.

This isn't just a matter of taste, for two reasons:

1. Interacting with objects via methods ensures that clients interact with objects *correctly*.

In [ ]:
```
# EXAMPLE 1a: Drawing cards, badly

import random
from evans_card_v1 import Card, Deck

d = Deck() # Create a deck.

# Pick 10 cards, in a flagrantly wrong way.
for i in range(10):
    card_num = random.randrange(len(d._cards)) # Look at the length of the deck, an
d pick an index in that range
    current_card = d._cards[card_num] # Pick the card in that index
    current_card.display()   # Display it
```

The problem here is that none of the code in the loop actually takes care of the part where cards are *removed* from the deck after they are picked.

Admittedly, this might seem like a pretty dumb mistake to make. But remember: as your classes get more and more complex, it is more and more difficult to avoid making similarly "dumb" mistakes. One way to avoid your users (or yourself) from making such mistakes is to make methods that act properly on objects, rather than having clients directly access the attribute variables (and risking mistakes creeping in) over and over.

2. Interacting with objects via methods ensures that your client code won't be broken by *implementation changes.*

Imagine that, after I had already started using my class, I decided to change how I represented a `Deck`. Instead of using a list of `Card`s which we shuffle at the time of initialization, suppose that we decided that it was smarter going forward to represent a `Deck` as a dictionary: the keys will be all the possible cards, and the values will be `True` if the card has already been drawn, and `False` if it has not been drawn (we'll talk about dictionaries in a minute).

Here's the point: *the lines of code below that **only reference methods** will still work if we use the updated class; the code that references attributes will break!*

In [ ]:
```
# EXAMPLE 1b: Changing Implementation
import random

# IN THE BEGINNING, ALL THE CODE WILL WORK.  HOWEVER, IF I CHANGE
# evans_card_v1 TO evans_card_v2
# SOME OF THE CODE WILL STOP WORKING.
from evans_card_v1 import Card, Deck


d = Deck() # Create a deck.

print("The good loop:\n")
for i in range(10): # This only uses methods.
    c = d.draw()    # It will NOT be broken by a change in implementation.
    c.display()     #

print("\nThe bad loop:\n")
for i in range(10):                             # This references the member variab
le ._cards
    card_num = random.randrange(len(d._cards))  # and it WILL be broken after a cha
nge in implementation.
    current_card = d._cards[card_num]           #
    del d._cards[card_num]                      #
    current_card.display()                      #
```

So, as a client, respecting the so-called "privacy" of attribute variables means that your code will be much less likely to break in the event of implementation change! And as a class designer, you can *refactor* your class safely so that it works better or more efficiently, as long as you don't change the way the interface works.

2. Dictionaries

The dictionary is a data structure that is meant to represent a *mapping* between "keys" (inputs) and "values" (outputs). A dictionary contains a number of key-value pairs. You may then use the dictionary by supplying it a key; the dictionary will supply the corresponding value.

The idea you should keep in mind is a translation dictionary. In this case, the keys would be English words, and the corresponding values would be the Spanish translations. When you perform a translation, you don't want to look up the position in the dictionary where your English word is located, and then separately look up the corresponding Spanish word -- you want to go straight from English word to Spanish word.

Let's see how they work.

In [ ]:
```
# EXAMPLE 2a: Dictionaries

# This is a dictionary.  It is enclosed in curly braces.  Each pair of values is se
parated by a comma,
# and in each pair, key and value are separated by a colon.
eng_spn = {"dog":"perro", "cat":"gato", "horse":"caballo"}

# You can access a dictionary by key, which looks a lot like accessing a list by in
dex
print(eng_spn["dog"])

word = input("Enter a word: ")
# One problem with dictionaries: you get an error if you try to look up a key that
is not present.
# For these cases: exceptions to the recsue!
try:
    trans = eng_spn[word]
    print(trans)
except KeyError:
    print("Why did you enter a word not in the dictionary?")

# Dictionaries are mutable.  You can add new entries like this:
eng_spn["cow"] = "waca"
# and you can change the values of each key after assignment:
eng_spn["cow"] = "vaca"

# You can also delete like you do with lists -- this will delete both key and value
.
del eng_spn["horse"]

# This is what it looks like when you print out a dictionary
print(eng_spn)
```

So, to summarize what the above code demonstrates:

- You can define a dictionary using curly braces.
- Each entry is of the form `key:value`, seperated by commas.
- You can access elements using *dictionary_name[key]*.
- It might be a good idea to surround code that accesses elements with try-except, since attempting to access an element with an invalid key will lead to an error.
- You can add new values using *dictionary_name[key] = value*.
- You can also modify existing values *dictionary_name[key] = value*, where *key* is a key that is already present.
- You can delete pairs using *del dictionary_name[key]*.

Let's try an example. Make a dictionary called `vote_counts`, containing three pairs: the values will be the strings `"Alice"`, `"Bob"`, and `"Carol"`, and the corresponding values will start with 0.

Inside the loop, have the program ask `"Who do you want to vote for?"`. The user will then type in a candidate they wish to vote for, and then the count for that candidate will be increased by 1. Can you make the code support write-in candidates?

```
In [ ]: # EXAMPLE 2b: Votes

        # Here's a dictionary called vote_counts
        vote_count = {}
        candidate_list = ["Alice", "Bob", "Carol"]
        for name in candidate_list:
            vote_count[name] = 0
        print(vote_count)


        # Now, add votes
        for i in range(5):
            vote = input("Who do you want to vote for? ")
            try:
                vote_count[vote] += 1
            except KeyError:
                # If vote is not one of the listed candidates,
                # add an entry to the dictionary, with its value equal to 1.
                vote_count[vote] = 1

        print(vote_count)
```

You can also loop through a dictionary. However, be aware that the entries in a dictionary *don't have a fixed order*. In an ordinary list, the order is given by the indices (0,1,2,....). In a dictionary, there are no indices -- only keys! (Admittedly, this example may not highlight this fact -- they'll probably print out in the order supplied.)

The loop will look like:

```
for k in dictionary:
```

The target variable (k) will take on all the *keys* in the dictionary; if you need to access the values, you can of course use `dictionary[k]`. Here's what I mean.

In [ ]:
```
# EXAMPLE 2c: Looping through dictionaries

vote_count = {"Alice":3, "Bob":1, "Carol":1}

# You can loop through a dictionary looks just like looping through a list.
for k in vote_count:
    # k is the KEY of each pair.
    print(k, "got", vote_count[k], "votes.")

#
# Now: write code that finds the winner of the election, using a loop. Assume no ti
es.
# It's actually a little tricky -- you might want to have two variables to keep tra
ck of (winner's name, winning count)
#
max_votes = 0
for k in vote_count:
    if vote_count[k] > max_votes:
        max_votes = vote_count[k]
        current_winner = k
print("Winner is:", current_winner)
```

If you just want to look at the keys of a dictionary, you can create a list out of them using `.keys()`:

`evans_new_list = list(dict_name.keys())`

And if you just want to look at the values of a dicitionary, you can similarly use `.values()`:

`evans_new_list = list(dict_name.values())`

In [ ]:
```
# EXAMPLE 2d: Another way to find the winner.
# This is a bit hacky.

vote_count = {"Alice":3, "Bob":1, "Carol":1}

# First, create the list of counts
count_list = list(vote_count.values())

# Then, find the max value.  We're getting to be grown ups now, so let's just use t
he max() function.
max_votes = max(count_list)

# Now, go through the dictionary and find the key whose value is the winning_count.
for k in vote_count:
    if vote_count[k] == max_votes:
        print("Winner is:", k)
```

3. In Which Evan Stops Withholding Lots of Python Tricks That Make Common Tasks Very Easy, Because Really, We're All Grown Ups and We Don't Need to Reinvent the Friggin Wheel Everyday

In the last section, we used the `max()` function to find the maximum of a list. Many you have no doubt already discovered this function a long time ago. It takes a list as input, and outputs the value of its maximum element.

Every programmer should have no trouble implementing this operation from scratch -- if you can't solve that problem, there's no way you'll be able to solve a non-trivial problem that doesn't have a canned solution. But maybe we're comfortable with the basic max-value-of-a-list problem by now.

Here are a few more shortcuts that I think you might want to know about.

In [ ]:
```
# EXAMPLE 3a: Some neat Python list tricks
a_list = [5, 18, 20, 4, -6, 37, 33, 45, 18]

# How do you find the min value of the list? Using min()
my_min_value = min(a_list)
print("The min value is:", my_min_value)

# How do you sum a list? Using sum()
my_sum = sum(a_list)
print("The sum is:", my_sum)

# This is a good one: how do you check if a number is contained in a list?  The "in
" keyword works well for that!
is_20_in = 20 in a_list
print("Is 20 in the list?", is_20_in)
is_30_in = 30 in a_list
print("Is 30 in the list?", is_30_in)

# How do you count how many times a number appears? Using .count()
num_of_18s = a_list.count(18)
num_of_157s = a_list.count(157)
print("There are {0} 18's and {1} 157's".format(num_of_18s, num_of_157s))

# How do you find the POSITION of a number in the list? Use .index() (whose behavio
r has a couple of subtleties)
where_is_20 = a_list.index(20)
where_is_18 = a_list.index(18) # It gives the FIRST occurence.
print("20 is at position {0}, 18 is at position {1}".format(where_is_20, where_is_1
8))
problem = a_list.index(157) # This will give an ERROR!
```

So, to summarize:

- `max()` and `min()` take lists as arguments, and return the max or min value of the list.
- `sum()` takes a list as an argument, and returns its sum
- `in`, which we've used with for loops, also acts as a membership test. So, e.g., you can write a line like `if x in some_list:`.
- `.count()`, when attached to a list, takes a value as argument, and returns the number of appearances of that value in the list.
- `.index()`, when attached to a list, takes a value as argument, and "finds" that value -- that is, if it is in the list, it will return the position of its first appearance.

This is not an exhaustive list. You're welcome to use them from now on unless I say otherwise.

4. List Comprehensions

One more cool thing that I'm just going to flash at you very quickly: the list comprehension. This is an amazing tool for constructing lists quickly.

In [ ]:
```
# EXAMPLE 4a: List comprehensions

numbers = input("Enter a list of numbers on one line: ")
# This of course gives a big string.  Let's say you want to turn this string
# into a list of floats. We know how to do that:

list_1 = []
for x in numbers.split():
    list_1.append(float(x))

# Here's an alternate way to write this code in one line:
# a list comprehension!
list_2 = [float(x) for x in numbers.split()]

print(list_1)
print(list_2)
```

The syntax for a list comprehension is:

In [ ]:
```
LIST COMPREHENSION SYNTAX:

<list name> = [<operation on item> for <item> in <list>]
```

This creates a list, where each value comes from performing the indicated operation on each element in `<list>`. For example, how can we create a new list which is equal to some old list with each element squared?

In [ ]:
```
# EXAMPLE 4b: Square the list

old = [5, 3, 8, 20, 17, 20, 64]
new = [x**2 for x in old]
print(new)

# And let's say you wanted to find the sum of the squares of elements in old.
# Can we do THAT in one line?
print(sum([x**2 for x in old]))
```

You can also use list comprehensions together with filters.

In [ ]:
```
LIST COMPREHENSION WITH FILTER SYNTAX:

<list name> = [<operation on item> for <item> in <list> if <condition on item>]
```

This will only add elements for `<item>`s which satisfy the `<condition>`. For example, let's do the problem from before, except now we'll make a list that contains only the squares of the elements that happen to be odd.

In [ ]:
```
# EXAMPLE 4c: Square the odd elements in the list

old = [5, 3, 8, 20, 17, 20, 64]
new = [x**2 for x in old if x%2 == 1]
print(new)
```

One more application: a program which converts a line of text to a list of numbers, but this time ignores non-numerical entries. It uses exceptions.

In [ ]:
```
# EXAMPLE 4d: Line of input to list of numbers:
# A SAFE version, which ignores non "floatable" entries.

def floatable(x):
    """Return true if the float function can be applied safely."""
    try:
        float(x)
        # If we make it past this line, we're safe.
        return True
    except ValueError:
        return False

numbers = input("Enter a list of numbers on one line: ")
# This simply ignores entries which can't be made into floats.
safe_list = [float(x) for x in numbers.split() if floatable(x)]

print(safe_list)
```