Course Name: Practical Python Programming

Week 3: Python Programing – Logic, List and more

a. Memorizing Logic.

We will start learning about logic. Up to this point, you have done everything you possibly can, reading and writing files to the Terminal, and have learned quite a lot of the math capabilities of Python.

The Truth Terms
In Python we have the following terms (characters and phrases) for determining if something is "True" or "False." Logic on a computer is all about seeing if some combination of these characters and some variables is True at that point in the program.

- **and**
- **or**
- **not**
- **!= (not equal)**
- **== (equal)**
- **>= (greater- than- equal)**
- **<= (less- than- equal)**
- **True**
- **False**

NOTE 1: We put a , (comma) at the end of each print line. This is so that print doesn't end the line with a new line character and go to the next line.

NOTE 2: Go on-line and find out what Python's raw_input does.

NOTE 3: Find other ways to use it? Try some of the samples you find.

NOTE 4: Write another “form” like this to ask some other questions.

NOTE 5: Related to escape sequences, try to find out why the last line has '6\'2"' with that \' sequence. See how the single-quote needs to be escaped because otherwise it would end the string?.

b. Boolean Practice.

The logic combinations you learned from the last exercise are called “boolean” logic expressions. Boolean logic is used everywhere in programming. They are essential fundamental parts of computation, an knowing them very well is akin to knowing your scales in music. In this exercise, you will take the logic exercises you memorized and start trying them out in Python. Take each of these logic problems, and write out what you think the answer will be. In each case, it will be either True or False. Once you have the answers written down, you will start Python in your Terminal and type them in to confirm your answers.

```
1. True and True
2. False and True
3. 1 == 1 and 2 == 1
4. "test" == "test"
5. 1 == 1 or 2 != 1
6. True and 1 == 1
7. False and 0 != 0
8. True or 1 == 1
9. "test" == "testing"
10. 1 != 0 and 2 == 1
11. "test" != "testing"
12. "test" == 1
13. not (True and False)
14. not (1 == 1 and 0 != 1)
15. not (10 == 1 or 1000 == 1000)
16. not (1 != 10 or 3 == 4)
17. not ("testing" == "testing" and "Zed" == "Cool Guy")
18. 1 == 1 and not ("testing" == 1 or 1 == 0)
19. "chunky" == "bacon" and not (3 == 4 or 3 == 3)
20. 3 == 3 and not ("testing" == "testing" or "Python" == "Fun")
```

One easy way to figure this out is: whenever you see these boolean logic statements, you can solve them easily by this simple process:

1. Find an equality test (== or !=) and replace it with its truth.
2. Find each and/or inside parentheses and solve those fi rst.
3. Find each not and invert it.
4. Find any remaining and/or and solve it.
5. When you are done, you should have True or False.

For example:

3 != 4 and not ("testing" != "test" or "Python" == "Python")

Here’s me going through each of the steps and showing you the translation until I’ve boiled it down to a single result:

1. Solve each equality test:

    a. 3 != 4 is True: True and not ("testing" != "test" or "Python" == "Python")
    b. "testing" != "test" is True: True and not (True or "Python" == "Python")
    c. "Python" == "Python": True and not (True or True)

2. Find each and/or in parentheses ():
    a. (True or True) is True: True and not (True)

3. Find each not and invert it:
    a. not (True) is False: True and False

4. Find any remaining and/or and solve them:
    a. True and False is False

With that, we're done and know the result is False.

NOTE 1. There are a lot of operators in Python similar to != and ==. Try to find out as many "equality operators" as you can. They should be like < or <=.

NOTE 2. Write out the names of each of these equality operators. For example, I call != "not equal."

NOTE 3. Play with the Python by typing out new boolean operators, and before you hit Enter, try to shout out what it is. Do not think about it—just name the first thing that comes to mind. Write it down, then hit Enter, and keep track of how many you get right and wrong.

NOTE 4. Throw away the piece of paper from #3 so you do not accidentally try to use it later.

c. What If.

This section introduces you to the if- statement. Type this in, make it run exactly right, and then we'll see if your practice has paid off.:

```
people = 20
cats = 30
dogs = 15

if people < cats:
      print "Too many cats! The world is doomed!"

if people > cats:
      print "Not many cats! The world is saved!"
if people < dogs:
      print "The world is drooled on!"

if people > dogs:
      print "The world is dry!"

dogs += 5
if people >= dogs:
      print "People are greater than or equal to dogs."

if people <= dogs:
      print "People are less than or equal to dogs."
```

```
if people == dogs:
      print "People are dogs."
```

NOTE 1. What do you think the if does to the code under it?

NOTE 2. Why does the code under the if need to be indented four spaces?

NOTE 3. What happens if it isn't indented?

NOTE 4. Can you put other boolean expressions from Exercise 27 in the if- statement? Try it.

NOTE 5. What happens if you change the initial variables for people, cats, and dogs?

d. Else and If.

In the last exercise, we worked out some if- statements and then tried to guess what they are and how they work. Before you learn more, Let us answer the following questions:

1. What do you think the if does to the code under it? An if- statement creates what is called a "branch" in the code. It's kind of like those choose-your-own-adventure books where you are asked to turn to one page if you make one choice and another if you go a different direction. The if- statement tells your script, "If this boolean expression is True, then run the code under it, otherwise skip it."

2. Why does the code under the if need to be indented four spaces? A colon at the end of a line is how you tell Python you are going to create a new "block" of code, and then indenting four spaces tells Python what lines of code are in that block. This is exactly the same thing you did when you made functions in the last week's class.

3. What happens if it isn't indented? If it isn't indented, you will most likely create a Python error. Python expects you to indent something after you end a line with a : (colon).

4. Can you put other boolean expressions from Exercise 27 in the if- statement? Try it.
Yes, you can, and they can be as complex as you like, although really complex things generally are bad style.

5. What happens if you change the initial values for people, cats, and dogs? Because you are comparing numbers, if you change the numbers, different if- statements will evaluate to True, and the blocks of code under them will run. Go back and put different numbers in and see if you can figure out in your head what blocks of code will run.

Type this one (use else .. if)) in and make it work too.

```
people = 30
cars = 40
buses = 15

if cars > people:
      print "We should take the cars."
elif cars < people:
      print "We should not take the cars."
else:
      print "We can't decide."
```

```
if buses > cars:
      print "That's too many buses."
elif buses < cars:
      print "Maybe we could take the buses."
else:
      print "We still can't decide."

if people > buses:
      print "Alright, let's just take the buses."
else:
      print "Fine, let's stay home then."
```

NOTE 1. Try multiple “elif”.

e. Making Decisions.

In the last script we wrote out a simple set of tests asking some questions. In this script we will ask the user questions and make decisions based on their answers. Write this script, and then play with it quite a lot to figure it out:

```
print "You enter a dark room with two doors. Do you go through door #1
or door #2?"

door = raw_input("> ")

if door == "1":
      print "There's a giant bear here eating a cheese cake. What do you
      do?"
      print "1. Take the cake."
      print "2. Scream at the bear."

      bear = raw_input("> ")

      if bear == "1":
            print "The bear eats your face off. Good job!"
      elif bear == "2":
            print "The bear eats your legs off. Good job!"
      else:
            print "Well, doing %s is probably better. Bear runs away." %
            bear
elif door == "2":
      print "You stare into the endless abyss at Cthulhu's retina."
      print "1. Blueberries."
      print "2. Yellow jacket clothespins."
      print "3. Understanding revolvers yelling melodies."

      insanity = raw_input("> ")

      if insanity == "1" or insanity == "2":
            print "Your body survives powered by a mind of jello. Good
job!"
      else:
            print "The insanity rots your eyes into a pool of muck. Good
job!"
```

```
else:
      print "You stumble around and fall on a knife and die. Good job!"
```

A key point here is that we are now putting the if- statements inside if- statements as code that can run. This is very powerful and can be used to create "nested" decisions, where one branch leads to another and another. Make sure you understand this concept of if- statements inside if- statements.

f. Loops And Lists.

We should now be able to do some programs that are much more interesting. If you have been keeping up, you should realize that now you can combine all the other things you have learned with if- statements and boolean expressions to make your programs do smart things. However, programs also need to do repetitive things very quickly. We are going to use a for-loop in this exercise to build and print various lists. When you do the exercise, you will start to figure out what they are. I won't tell you right now. You have to figure it out. Before you can use a for- loop, you need a way to store the results of loops somewhere. The best way to do this is with a list. A list is exactly what its name says—a container of things that are organized in order. It's not complicated; you just have to learn a new syntax. First, there's how you make a list:

```
hairs = ['brown', 'blond', 'red']
eyes = ['brown', 'blue', 'green']
weights = [1, 2, 3, 4]
```

What you do is start the list with the [(left bracket), which "opens" the list. Then you put each item you want in the list separated by commas, just like when you did function arguments. Lastly you end the list with a] (right bracket) to indicate that it's over. Python then takes this list and all its contents and assigns them to the variable.

We now will build some lists using some loops and print them out:

```
the_count = [1, 2, 3, 4, 5]
fruits = ['apples', 'oranges', 'pears', 'apricots']
change = [1, 'pennies', 2, 'dimes', 3, 'quarters']

# this first kind of for- loop goes through a list
for number in the_count:
      print "This is count %d" % number

# same as above
for fruit in fruits:
      print "A fruit of type: %s" % fruit

# also we can go through mixed lists too
# notice we have to use %r since we don't know what's in it
for i in change:
      print "I got %r" % i

# we can also build lists, first start with an empty one
elements = []

# then use the range function to do 0 to 5 counts
for i in range(0, 6):
      print "Adding %d to the list." % i
      # append is a function that lists understand
      elements.append(i)
```

```
# now we can print them out too
for i in elements:
      print "Element was: %d" % i
```

NOTE 1. Take a look at how we used range. Look up the range function to understand it.

NOTE 2. Could we have avoided that for- loop entirely on line 22 and just assigned range(0,6) directly to elements?

NOTE 3. Find the Python documentation on lists and read about them. What other operations can we do to lists besides append?

g. While-Loops.

Now we are going to learn a new loop—the while- loop. A while- loop will keep executing the code block under it as long as a boolean expression is True.

Wait, we have been keeping up with the terminology, right? That if we write a line and end it with a : (colon), then that tells Python to start a new block of code? Then we indent and that's the new code. This is all about structuring your programs so that Python knows what you mean. If you do not get that idea, then go back and do some more work with if- statements, functions, and the for- loop until you get it. Later on, we'll have some exercises that will train your brain to read these structures, similar to how we burned boolean expressions into our brain.

Back to while- loops. What they do is simply do a test like an if- statement, but instead of running the code block once, they jump back to the "top" where the while is and repeat. It keeps doing this until the expression is False.

Here's the problem with while- loops: Sometimes they do not stop. This is great if your intention is to just keep looping until the end of the universe. Otherwise you almost always want your loops to end eventually.

To avoid these problems, there's some rules to follow:

1. Make sure that you use while- loops sparingly. Usually a for- loop is better.
2. Review the while statements and make sure that the thing you are testing will become False at some point.
3. When in doubt, print out your test variable at the top and bottom of the while- loop to see what it's doing.

In this exercise, you will learn the while- loop by doing the above three things:

```
i = 0
numbers = []

while i < 6:
      print "At the top i is %d" % i
      numbers.append(i)

      i = i + 1
      print "Numbers now: ", numbers
      print "At the bottom i is %d" % i
print "The numbers: "

for num in numbers:
      print num
```

NOTE 1. Convert this while- loop to a function that you can call, and replace 6 in the test (i < 6) with a variable.

NOTE 2. Now use this function to rewrite the script to try different numbers.

NOTE 3. Add another variable to the function arguments that you can pass in that lets you change the + 1 on line 8, so you can change how much it increments by.

NOTE 4. Rewrite the script again to use this function to see what effect that has.

NOTE 5. Now, write it to use for- loops and range instead. Do you need the incrementor in the middle anymore? What happens if you do not get rid of it?

h. Accessing Elements of Lists.

Lists are pretty useful, but only if you can get at the things inside them. You can already go through the elements of a list in order, but what if you want, say, the fifth element? You need to know how to access the elements of a list. Here's how you would access the first element of a list:

```
animals = ['bear', 'tiger', 'penguin', 'zebra']
bear = animals[0]
```

You take a list of animals, and then you get the first (1st) one using 0?! How does that work? Because of the way math works, Python start its lists at 0 rather than 1. It seems weird, but there's many advantages to this, even though it is mostly arbitrary.

i. Branches and Functions.

You have learned to do if- statements, functions, and lists. Now it's time to bend your mind. Type this in, and see if you can figure out what it's doing.

```
from sys import exit

def gold_room():
      print "This room is full of gold. How much do you take?"
      next_move = raw_input("> ")
      if "0" in next_move or "1" in next_move:
            how_much = int(next_move)
      else:
            dead("Man, learn to type a number.")

      if how_much < 50:
            print "Nice, you're not greedy, you win!"
            exit(0)
      else:
            dead("You greedy bastard!")

def bear_room():
      print "There is a bear here."
      print "The bear has a bunch of honey."
      print "The fat bear is in front of another door."
      print "How are you going to move the bear?"
      bear_moved = False
      while True:
            next_move = raw_input("> ")
            if next_move == "take honey":
                  dead("The bear looks at you then slaps your face
off.")
            elif next_move == "taunt bear" and not bear_moved:
                  print "The bear has moved from the door. You can go
through it now."
                  bear_moved = True
            elif next_move == "taunt bear" and bear_moved:
                  dead("The bear gets pissed off and chews your leg
off.")
            elif next_move == "open door" and bear_moved:
                  gold_room()
            else:
                  print "I got no idea what that means."

def cthulhu_room():
      print "Here you see the great evil Cthulhu."
      print "He, it, whatever stares at you and you go insane."
      print "Do you flee for your life or eat your head?"

      next_move = raw_input("> ")

      if "flee" in next_move:
            start()
      elif "head" in next_move:
            dead("Well that was tasty!")
      else:
            cthulhu_room()

def dead(why):
      print why, "Good job!"
      exit(0)

def start():
      print "You are in a dark room."
      print "There is a door to your right and left."
      print "Which one do you take?"
      next_move = raw_input("> ")
      if next_move == "left":
            bear_room()
      elif next_move == "right":
            cthulhu_room()
      else:
            dead("You stumble around the room until you starve.")

start()
```

QUESTION 1: **Why are you doing while True:?**
That makes an infinite loop.

QUESTION 2: **What does exit(0) do?**
On many operating systems, a program can abort with exit(0), and the number passed in will indicate an error or not. If you do exit(1), then it will be an error, but exit(0) will be a good exit. The reason it's backward from normal boolean logic (with 0==False) is that you can use different numbers to indicate different error results. You can do exit(100) for a different error result than exit(2) or exit(1).

QUESTION 3: **Why is raw_input() sometimes written as raw_input('> ')?**
The parameter to raw_input is a string that it should print as a prompt before getting the user's input.

j. Designing and Debugging.

Now that we know if- statements, I'm going to give you some rules for for- loops and while- loops that will keep us out of trouble. I'm also going to show you some tips on debugging so that we can figure out problems with our program.

Rules for If- Statements

1. Every if- statement must have an else.
2. If this else should never be run because it doesn't make sense, then you must use a die function in the else that prints out an error message and dies, just like we did in the last exercise. This will find many errors.
3. Never nest if- statements more than two deep and always try to do them one deep. This means if you put an if in an if, then you should be looking to move that second if into another function.
4. Treat if- statements like paragraphs, where each if, elif, else grouping is like a set of sentences. Put blank lines before and after.
5. Your boolean tests should be simple. If they are complex, move their calculations to variables earlier in your function and use a good name for the variable.

If you follow these simple rules, you will start writing better Python code.

Rules for Loops
1. Use a while- loop only to loop forever, and that means probably never. This only applies to Python; other languages are different.
2. Use a for- loop for all other kinds of looping, especially if there is a fixed or limited number of things to loop over.

k. Symbol Review

It's time to review the symbols and Python words you know and try to pick up a few more for the next few lessons. What I've done here is written out all the Python symbols and keywords that are important to know.

In this lesson take each keyword and fi rst try to write out what it does from memory. Next, search online for it and see what it really does. It may be hard because some of these are going to be impossible to search for, but keep trying.

If you get one of these wrong from memory, write up an index card with the correct definition, and try to "correct" your memory. If you just didn't know about it, write it down, and save it for later.

Finally, use each of these in a small Python program, or as many as you can get done. The key here is to find out what the symbol does, make sure you got it right, correct it if you do not, then use it to lock it in.

Keywords

- **and**
- **del**
- **from**
- **not**
- **while**
- **as**
- **elif**
- **global**
- **or**
- **with**
- **assert**
- **else**
- **if**
- **pass**
- **yield**
- **break**
- **except**
- **import**
- **print**
- **class**
- **exec**
- **in**
- **raise**
- **continue**
- **finally**
- **is**
- **return**
- **def**
- **for**
- **lambda**
- **try**

Data Types

For data types, write out what makes up each one. For example, with strings write out how you create a string. For numbers, write out a few numbers.

- **True**
- **False**
- **None**
- **strings**
- **numbers**
- **floats**
- **lists**

String Escape Sequences

For string escape sequences, use them in strings to make sure they do what you think they do.

- \\
- \'
- \"
- \a
- \b
- \f
- \n
- \r
- \t
- \v

String Formats

Same thing for string formats: use them in some strings to know what they do.

- %d
- %i
- %o
- %u
- %x
- %X
- %e
- %E
- %f
- %F
- %g
- %G
- %c
- %r
- %s
- %%

Operators

Some of these may be unfamiliar to you, but look them up anyway. Find out what they do, and if you still can't fi gure it out, save it for later.

- +
- -
- *
- **
- /
- //
- %
- <
- >
- <=
- >=
- ==
- !=
- <>
- ( )

- **[ ]**
- **{ }**
- **@**
- **,**
- **:**
- **.**
- **=**
- **;**
- **+=**
- **- =**
- ***=**
- **/=**
- **//=**
- **%=**
- ****=**

QUESTION 1: **Functions and what they do?**

QUESTION 2: **Where each variable is first given a value?**

QUESTION 3: **Can variables with the same names exist in different parts of the same program?**

QUESTION 4: **Any while- loops that might not end?**

QUESTION 5: **Any if- statements without else clauses. Are they right?**

QUESTION 6: **What's the difference between %d and %i formatting?**
Shouldn't be any difference, other than the fact that people use %d more due to historical reasons.

I. Doing Things to Lists

We have learned about lists. When we learned about while- loops, we "appended" numbers to the end of a list and printed them out. When we type Python code that reads mystuff.append('hello'), we are actually setting off a chain of events inside Python to cause something to happen to the mystuff list. Here's how it works:

1. Python sees we mentioned mystuff and looks up that variable. It might have to look backward to see if we created it with =, and look and see if it is a function argument or a global variable. Either way, it has to fi nd the mystuff first.
2. Once it finds mystuff it then hits the . (period) operator and starts to look at variables that are a part of mystuff. Since mystuff is a list, it knows that mystuff has a bunch of functions.
3. It then hits append and compares the name "append" to all the ones that mystuff says it owns. If append is in there (it is), then it grabs that to use.
4. Next Python sees the ((parenthesis) and realizes, "Oh hey, this should be a function." At this point it calls (a.k.a. runs, executes) the function just like normally, but instead it calls the function with an extra argument.
5. That extra argument is . . . mystuff! I know, weird, right? But that's how Python works so it's best to just remember it and assume that's alright. What happens then, at the end of all this, is a function call that looks like append(mystuff, 'hello') instead of what we read, which is mystuff.append('hello').

That might be a lot to take in, but we're going to spend a few exercises getting this concept firm in your brain. To kick things off, here's an exercise that mixes strings and lists for all kinds of fun.

```
ten_things = "Apples Oranges Crows Telephone Light Sugar"

print "Wait there's not 10 things in that list, let's fix that."

stuff = ten_things.split(' ')
more_stuff = ["Day", "Night", "Song", "Frisbee", "Corn", "Banana",
"Girl", "Boy"]

while len(stuff) != 10:
      next_one = more_stuff.pop()
      print "Adding: ", next_one
      stuff.append(next_one)
      print "There's %d items now." % len(stuff)

print "There we go: ", stuff

print "Let's do some things with stuff."

print stuff[1]
print stuff[- 1] # whoa! fancy
print stuff.pop()
print ' '.join(stuff) # what? cool!
print '#'.join(stuff[3:5]) # super stellar!

print "We can also do that this way:"
print "We'd have %d beans, %d jars, and %d crates." % secret_formu-
la(start_point)
```

NOTE 1. Take each function that is called, and go through the steps outlined above to translate them to what Python does. For example, ' '.join(things) is join(' ', things).

NOTE 2. Translate these two ways to view the function calls. For example, ' '.join(things) reads as, “Join things with ‘ ‘ between them.” Meanwhile, join(' ', things) means, “Call join with ‘ ‘ and things.” Understand how they are really the same thing.

NOTE 3. Go read about “object- oriented programming” online. Confused? I was too. Do not worry. You will learn enough to be dangerous, and you can slowly learn more later.

NOTE 4. Read up on what a “class” is in Python. Do not read about how other languages use the word “class.” That will only mess you up.

NOTE 5. What’s the relationship between dir(something) and the “class” of something?

NOTE 6. If you do not have any idea what I’m talking about, do not worry. Programmers like to feel smart, so they invented object- oriented programming, named it OOP, and then used it way too much. If you think that’s hard, you should try to use “functional programming.”

m. Dictionaries, Oh Lovely Dictionaries

Now I have to hurt you with another container you can use, because once you learn this container, a massive world of ultra- cool will be yours. It is the most useful container ever: the dictionary.

Python calls them “dicts.” Other languages call them “hashes.” I tend to use both names, but it doesn’t matter. What does matter is what they do when compared to lists. You see, a list lets you do this:

```
>>> things = ['a', 'b', 'c', 'd']
>>> print things[1]
b
>>> things[1] = 'z'
>>> print things[1]
z
>>> print things
['a', 'z', 'c', 'd']
>>>
```

You can use numbers to “index” into a list, meaning you can use numbers to find out what’s in lists. You should know this about lists by now, but make sure you understand that you can only use numbers to get items out of a list.

What a dict does is let you use anything, not just numbers. Yes, a dict associates one thing to another, no matter what it is. Take a look:

```
>>> stuff = {'name': 'Zed', 'age': 36, 'height': 6*12+2}
>>> print stuff['name']
Zed
>>> print stuff['age']
36
>>> print stuff['height']
74
>>> stuff['city'] = "San Francisco"
>>> print stuff['city']
San Francisco
>>>
```

You will see that instead of just numbers we’re using strings to say what we want from the stuff dictionary. We can also put new things into the dictionary with strings. It doesn’t have to be strings though. We can also do this:

```
>>> stuff[1] = "Wow"
>>> stuff[2] = "Neato"
>>> print stuff[1]
Wow
>>> print stuff[2]
Neato
>>> print stuff
{'city': 'San Francisco', 2: 'Neato',
'name': 'Zed', 1: 'Wow', 'age': 36,
'height': 74}
>>>
```

In this code I used numbers, and then you can see there are numbers and strings as keys in the dict when I print it. I could use anything—well, almost, but just pretend you can use anything for now.

Of course, a dictionary that you can only put things in is pretty stupid, so here’s how you delete things, with the del keyword:

```
>>> del stuff['city']
>>> del stuff[1]
>>> del stuff[2]
>>> stuff
```

```
{'name': 'Zed', 'age': 36, 'height': 74}
>>>
```

We'll now do an exercise that all of us must study very carefully. Please type this exercise in and try to understand what's going on. Take note of when I put things in a dict, get from them, and all the operations I use here.

```
# create a mapping of state to abbreviation
states = {  'Oregon': 'OR',
            'Florida': 'FL',
            'California': 'CA',
            'New York': 'NY',
            'Michigan': 'MI'
         }

# create a basic set of states and some cities in them
cities = { 'CA': 'San Francisco',
           'MI': 'Detroit',
           'FL': 'Jacksonville'
         }

# add some more cities
cities['NY'] = 'New York'
cities['OR'] = 'Portland'

# print out some cities
print '- ' * 10
print "NY State has: ", cities['NY']
print "OR State has: ", cities['OR']

# print some states
print '- ' * 10
print "Michigan's abbreviation is: ", states['Michigan']
print "Florida's abbreviation is: ", states['Florida']

# do it by using the state then cities dict
print '- ' * 10
print "Michigan has: ", cities[states['Michigan']]
print "Florida has: ", cities[states['Florida']]

# print every state abbreviation
print '- ' * 10
for state, abbrev in states.items():
    print "%s is abbreviated %s" % (state, abbrev)

# print every city in state
print '- ' * 10
for abbrev, city in cities.items():
    print "%s has the city %s" % (abbrev, city)

# now do both at the same time
print '- ' * 10
for state, abbrev in states.items():
    print "%s state is abbreviated %s and has city %s" % (state, abbrev,
cities[abbrev])

print '- ' * 10
```

```
# safely get an abbreviation by state that might not be there
state = states.get('Texas', None)
if not state:
    print "Sorry, no Texas."

# get a city with a default value
city = cities.get('TX', 'Does Not Exist')
print "The city for the state 'TX' is: %s" % city
```

QUESTION 1: **What the difference between a list and a dictionary?**
A list is for an ordered list of items. A dictionary (or dict) is for matching some items (called “keys”) to other items (called “values”).

QUESTION 2: **What would I use a dictionary for?**
Use it any time you have to take one value and “look up” another value. In fact, you could call dictionaries “look up tables.”

QUESTION 3: **What would I use a list for?**
Use it any time you have to take one value and “look up” another value. In fact, you could call dictionaries “look up tables.”

QUESTION 4: **What if I need a dictionary, but I need it to be in order?**
Take a look at the collections.OrderedDict data structure in Python. Search for it on-line to find the documentation.