1. We saw that dictionaries are *unordered* collections. Write a `for` loop which prints a dictionary's items in sorted (ascending) order. Hint: use the dictionary `keys` and list `sort` methods.

2. Part of learning to program in Python is learning which coding alternatives work better than others. Consider the following code, which uses a `while` loop and `found` flag to search a list of powers-of-2, for the value of 2 raised to the power 5 (32). It's stored in a module file called `power.py`.

```
L = [1, 2, 4, 8, 16, 32, 64]
X = 5

found = i = 0
while not found and i < len(L):
    if 2 ** X == L[i]:
        found = 1
    else:
        i = i+1

if found:
    print 'at index', i
else:
    print X, 'not found'
```

```
C:\code> python power.py      # run this in a system prompt
```

As is, the example doesn't follow normal Python coding techniques. Follow the steps below to improve it; for all of the transformations, you may type your code interactively, or store it in a script file run from the system command line (though using a file will make this exercise much easier).

a) First, rewrite this code with a while loop `else`, to eliminate the `found` flag and final `if` statement.

b) Next, rewrite the example to use a `for` loop with an `else`, to eliminate the explicit list indexing logic. Hint: to get the index of an item, use the list *index* method (`L.index(X)` returns the offset of the first `X` in list `L`).

c) Now, remove the loop completely by rewriting the examples with a simple `in` operator membership expression (to see how, interactively type this: `2 in [1,2,3]`).

d) Finally, use a `for` loop and the list `append` method to generate the powers-of-2 list (`L`), instead of hard-coding a list constant.

3. Write a function called `adder` in a Python module file. `adder` should accept two arguments, and return the sum (or concatenation) of its two arguments. Then add code at the bottom of the file to call the function with a variety of object types (two strings, two lists, two floating-points), and run this file as a script from the system command line. Do you have to print the call statement results to see results on the screen?

4. Generalize the `adder` function you wrote in the last exercise to compute the sum of an arbitrary number of arguments, and change the calls to pass more or less than two. What type is the return value sum? (Hints: a slice like `S[:0]` returns an empty sequence of the same type as `S`, and the `type` built-in function can be used to test types). What happens if you pass in arguments of different types? What about passing in dictionaries?

5. Change the `adder` function from exercise (3) to accept and add three arguments: "`def adder(good, bad, ugly)`". Now, provide default values for each argument, and experiment with calling the function interactively. Try passing 1, 2, 3, and 4 arguments. Then, try passing keyword arguments. Does the call "`adder(ugly=1, good=2)`" work? Why? Finally, generalize the new `adder` to accept and add an arbitrary number of keyword arguments, much like exercise (3), but you'll need to iterate over a dictionary, not a tuple (hint: the `dictionary.keys()` method returns a list you can step through with a `for` or `while`).

6. Write a function called `copyDict(dict)`, which copies its dictionary argument. It should return a new dictionary with all the items in its argument. Use the dictionary `keys` method to iterate. Copying sequences is easy (`X[:]` makes a top-level copy); does this work for dictionaries too?

7. Write a function called `addDict(dict1, dict2)` which computes the union of two dictionaries. It should return a new dictionary, with all the items in both its arguments (assumed to be dictionaries). If the same key appears in both arguments, feel free to pick a value from either. Test your function by writing it in a file and running the file as a script. What happens of you pass lists instead of dictionaries? How could you generalize your function to handle this case too? (Hint: see the `type` built-in function used earlier). Does the order of arguments passed matter?

8. First, define the following six functions (either interactively, or in an importable module file):

```
def f1(a, b): print a, b             # normal args

def f2(a, *b): print a, b            # positional varargs

def f3(a, **b): print a, b           # keyword varargs

def f4(a, *b, **c): print a, b, c    # mixed modes

def f5(a, b=2, c=3): print a, b, c   # defaults

def f6(a, b=2, *c): print a, b, c    # defaults + positional varargs
```

Now, test the following calls interactively, and try to explain each result; in some cases, you'll probably need to fall back on the matching algorithm shown in the lecture. Do you think mixing matching modes is a good idea in general? Can you think of cases where it would be useful anyhow?

```
>>> f1(1, 2)
>>> f1(b=2, a=1)

>>> f2(1, 2, 3)
>>> f3(1, x=2, y=3)
>>> f4(1, 2, 3, x=2, y=3)

>>> f5(1)
>>> f5(1, 4)

>>> f6(1)
>>> f6(1, 3, 4)
```

9. Write code to build a new list containing the square roots of all the numbers in this list: [2, 4, 9, 16, 25]. Code this as a `for` loop first, then as a `map` call, and finally as a list comprehension. Use the `sqrt` function in the builting `math` module to do the calculation (i.e., `import math`, and say `math.sqrt(x)`). Of the three, which approach do you like best?

10. Write a program that counts lines and characters in a file (similar in spirit to "wc" on Unix). With your favorite text editor, code a Python module called `mymod.py`, which exports three top-level names:

    a) A `countLines(name)` function that reads an input file and counts the number of lines in it (hint: `file.readlines()` does most of the work for you, and `len` does the rest)

    b) A `countChars(name)` function that reads an input file and counts the number of characters in it (hint: `file.read()` returns a single string)

c) A `test(name)` function that calls both counting functions with a given input filename. Such a filename generally might be passed-in, hard-coded, input with `input`, or pulled from a command-line via the `sys.argv` list; for now, assume it's a passed-in function argument.

All three `mymod` functions should expect a filename string to be passed in. If you type more than two or three lines per function, you're working much too hard—use the hints listed above!

Now, test your module interactively, using import and name qualification to fetch your exports. Does your `PYTHONPATH` need to include the directory where you created `mymod.py`? Try running your module on itself: e.g., `test("mymod.py")`. Note that `test` opens the file twice; if you're feeling ambitious, you may be able to improve this by passing an open file object into the two count functions (hint: `file.seek(0)` is a file rewind).

11. Test your `mymod` module from Exercise 10 interactively, by using `from` to load the exports directly, first by name, then using the `from*` variant to fetch everything.

12. Now, add a line in your `mymod` module that calls the `test` function automatically only when the module is run as a script, not when it is imported The line you add will probably test the value of `__name__` for the string "`__main__`", as shown in this unit. Try running your module from the system command line; then, import the module and test its functions interactively. Does it still work in both modes?

13. Write a second module, `myclient.py`, which imports `mymod` and tests its functions; run `myclient` from the system command line. If `myclient` uses `from` to fetch from `mymod`, will `mymod`'s functions be accessible from the top level of `myclient`? What if it imports with `import` instead? Try coding both variations in `myclient` and test interactively, by importing `myclient` and inspecting its `__dict__`.

14. Simulate a fast-food ordering scenario by defining four classes:

a) `Lunch`: a container and controller class

b) `Customer`: the actor that buys food

c) `Employee`: the actor that a customer orders from

d) `Food`: what the customer buys

To get you started, here are the classes and methods you'll be defining:

```
class Lunch:
    def __init__(self)          # make/embed Customer and Employee
    def order(self, foodName)   # start a Customer order simulation
    def result(self)            # ask the Customer what kind of Food it has

class Customer:
    def __init__(self)                          # initialize my food to None
    def placeOrder(self, foodName, employee)    # place order with an Employee
    def printFood(self)                         # print the name of my food

class Employee:
    def takeOrder(self, foodName)     # return a Food, with requested name

class Food:
    def __init__(self, name)          # store food name
```

The order simulation works as follows:

a) The `Lunch` class's constructor should make and embed an instance of `Customer` and `Employee`, and export a method called `order`. When called, this `order` method should ask the `Customer` to place an order, by calling its `placeOrder` method. The `Customer`'s `placeOrder` method should in turn ask the `Employee` object for a new `Food` object, by calling the `Employee`'s `takeOrder` method.

b) Food objects should store a food name string (e.g., "burritos"), passed down from `Lunch.order` to `Customer.placeOrder`, to `Employee.takeOrder`, and finally to `Food`'s constructor. The top-level `Lunch` class should also export a method called `result`, which asks the customer to print the name of the food it received from the `Employee` via the order (this can be used to test your simulation).

c) Note that `Lunch` needs to either pass the `Employee` to the `Customer`, or pass itself to the `Customer`, in order to allow the `Customer` to call `Employee` methods.

Experiment with your classes interactively by importing the `Lunch` class, calling its order method to run an interaction, and then calling its result method to verify that the `Customer` got what he or she ordered. If you prefer, you can also simply code test cases as self-test code in the file where your classes are defined, using the module `__name__` trick we met in the modules unit. In this simulation, the `Customer` is the active agent; how would your classes change if `Employee` were the object that initiated customer/employee interaction instead?