Python Review (everything you should have learned in ICS-31/32)

When reading the following material, I suggest that you have Eclipse open, including a project with an empty module; then copy/paste some of the code below into it, to see what it does. Explore the code by experimenting with (changing) it and predicting what results from the changes. I always have an Eclipse folder/module named "experiment" open for this purpose. I believe this approach is superior to me typing/executing code during lecture.

This lecture note is long (it is really three lectures), but the information is critical for getting started. I hope that this is mostly a review for ICS-31/32 students, but there are likely to be many things that come up as new (or as new perspectives and connections to the material that you already know). Pay close attention to the terminology used here, as I will use it (I hope consistently) throughout the entire quarter. If you do not know any of these technical terms, try looking them up online, or post a message in the "Lecture Material" Piazza Q&A Folder: if you didn't understand a term, probably other students didn't as well. Here are 3 quotes relevant to this lecture:

1) The first step towards wisdom is calling things by their right names.

2) He/She who is ashamed of asking is ashamed of learning.

3) The voyage of discovery is not in seeking new landscapes but in having new eyes. - M. Proust

---

Python in Four Sentences:

1. Names (in namespaces) are bound to objects.

2. Everything that Python computes with is an object.
   (examples are instance/data, function, module, and class objects)

3. Every object has its own namespace.
   (a dictionary that binds its internal names to other objects)

4. Python has rules about how things work.

In some sense these four sentences tell you very little about Python, but in another sense they are a tiny framework in which to interpret every idea -small and big- in Python and how Python works.

Every name appears in the namespace of some object (when we define names in a module, for example, these names appear in the module object's namespace); and every name is itself bound to some object (names are bound when defined on the left of the = symbol; they can be rebound to another object by using them on the left of the = symbol again; names are also bound in import statements, function definitions, and class definitions: these names/bindings are discussed in ICS-33 throughout the quarter).

Objects are the fundamental unit with which Python computes. For example...

1) We can compute with int objects (instance objects from the int class) by using operators; for the int object bound to x by x = 1 we can rebind it to another int object, one bigger, by writing x = x + 1. We will learn later that Python translates x+1 into the method call x.__add__(1) and then into int.__add__(x,1) -assuming, as it does here, x stores a reference to an int object: int is a reference to a class object.

2) We can compute with function objects by calling them; for the function object bound to print, we can write print(x); we will learn later in these notes that we can also pass functions as arguments to functions and return functions as results from functions. Python allows us to do many interesting things with functions, beyond just calling them.

3) We can compute with module objects by importing them (and/or the objects bound to the names in their namespaces).

```
import random                          or     from random import randint
x = random.randint(1,6)                       x = randint(1,6)

The name random (in the current               The name randint (in the current
module) is bound to the random module         module) is bound to the function
object, which defines a randint               object that the name randint (in
function in its namespace, which is           the random module's namespace)
called.                                       is bound to, which is called.
```

4) We can compute with class objects by constructing instance objects and using the instances to call class methods. For example, we can write
timer = Stopwatch()
timer.start()

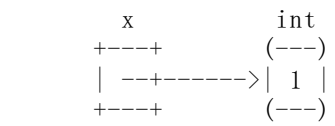
Python follows rules that determine how names are bound (e.g., how arguments are bound to parameters), how operators compute, how control structures execute the code blocks they control, etc. This course is designed to demystify how Python executes scripts and how you can better use its features to write scripts. We know a programming language in most part by knowing its rules.

Binding (and Drawing Names and their associated Objects)

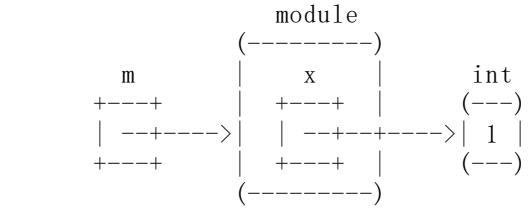
The process of making a name refer to a value: e.g., x = 1 binds the name x to the value 1 (which is an object/instance of the int class); later we can bind x to another value (possibly from another class) in another assignment: e.g., x = 'abc'. We speak about "the binding (noun) of a name" to mean the value (such values are always objects) that the name is currently associated with (or the object the name refers to).

In Python, every data instance, module, function, and class is an object that has a dictionary that stores its namespace: all its internal bindings. We will learn much more about namespaces (and how to manipulate them) later in the quarter, when we study classes in more detail.

Typically we illustrate the binding of a name to an object (below, x = 1) as follows. We write the name over a rectangle, in which the tail of an arrow is INSIDE, specifying the current reference stored for that name: the arrow's head refers to a rounded-edge rectangle object labeled by its type (here the class it is constructed from) and its value (inside).



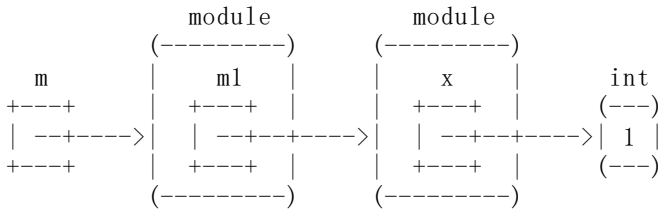
Technically, if we we write x = 1 inside the module m, Python has already created an object for module m (we show all objects as rounded-edge rectangles) and it puts x, its box, and its binding in the namespace of module m: here, the name x is defined inside module m and bound to object 1. That is, we would more formally write the result of x = 1 in module m as



But often we revert to the previous simpler diagram, when we don't care in what module x is defined, focusing solely on x and the object it refers to.

Note that the del command in Python (e.g., del name) removes a name from the current namespace/dictionary of the object in which name is bound. So, writing del x inside module m would remove x and its box from m's namspace/dictionary. If there were no name x in this module, Python raises an NameError exception.

More generally, if m1 (a name defined in module m) were to refer to another module that defined name x (see the picture below)



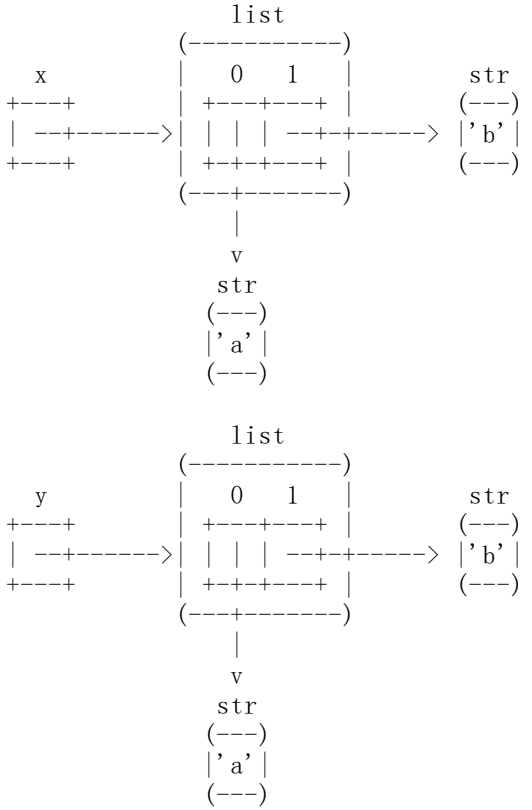
then writing del m1.x inside module m would remove x and its box from m1's namspace/dictionary. If there were no name x in this module, Python raises an NameError exception.

Finally, the "is" operator in Python determines whether two references refer to the same object (it is called the (object)identity operator); the == operator determines whether two references refer to objects that store the same internal state. If a is b is True then a == b must be True (because both a and b refer to the same object if a is b is True, and every object has the same state as itself.

If we write

```
x = ['a','b']
y = ['a','b']
```

then x is y is False and x == y is True: the names x and y are bound to/refer to two different list objects (each use of [] creates a new list), but these two objects store the same state ('a' at index 0; 'b' at index 1).



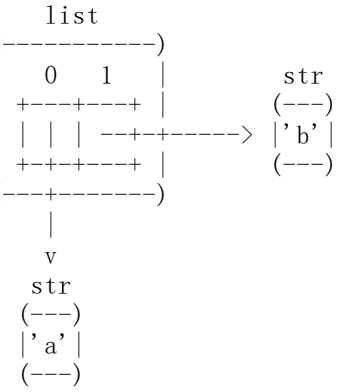
Technically, the 0 and 1 indexes of the list should share the same string objects ('a' and 'b'). This a subtle point that is special to int and str objects. I won't care whether you share or duplicate such objects. It is intersting to use is to tell what is actually happening:

```
print(x[0] is y[0], x[1] is y[1], x[0] == y[0], x[1] == y[1])
```

Likewise, if we write

```
  x = ['a','b']
  y = x
```

then we create one list object, and bind it to both x and y (assignment y = x just copies into name y the reference in name x, making y and x refer to the same object). Here, x is y is True (and x == y is therefore True): the names x and y refer to the same list object. We would diagram it as



So, pictures can help us understand the differences between these two code fragments. You should be able to draw a simple picture of these names and objects (both the list and int objects) to illustrate the difference between the "is" and == operators.

The == operator, especially in simple programs, is used much more frequntly than the the "is" operator.

What happens in each example (what picture results) if we execute y[0] = 'c'? What would be printed by print(x[0]) in each after such an assignment?

Finally, if we are in module m1 and we write

```
import m2
```

then we can create/delete names in m2's namespace by writing

```
m2.x = 3  # create a name x in the namespace of module m2 (present? use that
          # name); bind it to int 3

del m2.x  # delete name x in the namespace of module m2; absent? raise exception
```

Statements vs Expressions

The basic unit of speech in English is a sentence: sentences are built from noun and verb phrases. In Python the basic unit of code is a statement: statements are built from expressions (like a boolean expression in an if statement) and other statements (like block statements inside an if statement).

Statements are executed to cause an effect (e.g., binding/rebinding a name or producing output). Expressions are evaluated to compute a result (e.g., computing some formula, numeric, string, boolean, list, etc.). For example, the statement x = 1, when executed, causes a binding of the name x to an object representing the integer 1. The expression x+1, when evaluated, computes the object representing the integer 2. Typically we write expressions inside statements: two examples are x = x+1 and print(x+1): both "do something" with the computed value x+1 (the first rebinds x to it; the second prints it). The distinction between statements and expressions is important in most programming languages. Learn the technical meaning of these terms and be able to recognize statements and expressions in statements.

The print function is a bit strange. We call if for an effect (putting characters in the Console window) but like every function, also returns a value: when print is called successfully (doesn't raise any exceptions) it returns the value None. We typically don't do anything with this value. So,

we can write the code line

```
print(x)
```

which calls the print function (printing something in the Console window) and doing nothing with the None value print returns.

Control structures (even simple sequential ones, like blocks) are considered to be statements in Python. Python has rules describing how to execute statements. Control structures might contain both statements and expressions. The following is a conditional statement using if/else

```
if x == None:
    y = 0
else:
    y = 1
```

This if statement contains the expression x == None and the statements y = 0 and y = 1. Technically, x, None, 0 and 1 are "trivial" expressions (which trivially compute the object to which x is currently bound, and the object values None, 0, and 1 which are literals: preconstructed objects "named" by None, 0, and 1).

The following statement includes a conditional expression that binds a value to y: the conditional expression includes the expressions 0 (yes, an object by itself, or just a name refering to an object, is a simple expression), x == None, and 1.

```
y = (0 if x == None else 1)
```

We will discuss conditional statements vs. conditional expressions in more detail later in this lecture note.

None:

None is a value (object/instance) of NoneType; it is the only value of that type. Sometimes we use None as a default value for a parameter's argument; sometimes we use it as a return value of a function: in fact, a Python function that terminates without executing a return statement automatically returns the value None. If None shows up as an unexpected result printed in your code or more likely in a raised exception, look for some function whose return value you FORGOT to specify explicitly (or whose return statement somehow didn't get executed before Python executes the last statement in a function).

---

pass:

pass is the simplest statement in Python; semantically, it means "do nothing". Sometimes when we write statements, we aren't sure exactly which ones to write, so we might use pass as a placeholder until we write the actual statement we need. Often, in tiny examples, we use pass as the meaning of a function.

```
def f(x) : pass     or     def f(x) :
                               pass
```

If you are ever tempted to write the first if statement below, don't; instead write the second one, which is simpler: but, they produce equivalent results.

```
if a in s:                    # DON'T write this code
    pass
else:
    print(a,'is not in',s)


if a not in s:                # Write this code instead; it is equivalent
    print(a,'is not in',s)    # and simpler; strive to use Python simply
```

Importing: 5 Forms

There are five import forms; you should know what each does, and start to think about which is appropriate to use in any given situation. Note that in EBNF (a notation used to describe programming languages, which we will discuss soon) [...] means option and {...} means repeat 0 or more times, although this second form is sometimes written (...)* when describing the syntax of Python.

Fundamentally, import statements bind names to objects (one or both of which come from the imported module).

"import module-name" form: one or more modules, optionally renaming each
1. import module-name{,module-name}
2. import module-name [as alt-name] {,module-name [as alt-name]}

"from module-name import" form: one or mor attributes, optionally renaming each
3. from module-name import attr-name{,attr-name}
4. from module-name import attr-name [as alt-name] {,attr-name [as alt-name]}
5. from module-name import *

Above, alt-name is an alternative name to be bound to the imported object; attr-name is an attribute name already defined in the namespace of the module from which attr-name is imported.

The "import module-name" forms import the names of modules (not their attribute names). (1) bind each module-name to the object representing that imported module-name. (2) bind each alt-name to the object representing its preceding imported module-name. Using a module name, we can refer to its attributes using periods, such as by module-name.attribute-name

The "from module-name import" forms don't import a module-name, but instead import some/all attribute names defined/bound inside module-name. (3) bind each attr-name to the object bound to that attr-name in module-name. (4) bind each alt-name to the object bound to the preceding attr-name in module-name. (5) bind each name that is bound in module-name to the same object it is bound to in module-name.

Import (like an assignment, and a def, and a class definition) creates a name (if that name is not already in the namespace) and binds it to an object: the "import module-name" form binds names to module objects; the "from module-name import" form binds names to objects (instances, functions, modules, classes, etc.) defined inside modules (so now two modules contain names -maybe the same, maybe different, depending on which of form 3 or 4 is use- bound to the same objects).

The key idea as to which kind of import to use is to make it easy to refer to a name but not pollute the name space of the module doing the importing with too many names (which might conflict with other names in that module).

If a lot of different names in the imported module are to be used, or we want to make it clear when an attribute name in another module is used, use the "import module-name" form (1) and then qualify the names when used: for example

```
import random
# use:  random.choice(...)  and  random.randint(...)
```

If the imported module-name is too large for using repeatedly, use an abbreviaton by importing using an alt-name (2) : for example

```
import high_precision_math as hp_math
# use:  hp_math.sqrt(...)
```

If only one name (or a few names) are to be used from a module, use the form (3):

```
from random import choice, randint
# use: choice(...)  and  randint(...)
```

Again, use alt-name to simplify either form, if the name is too large and unwieldy to use. Such names are often very long to be clear, but awkward to use many times at that length. Generally we should apply the Goldilocks principle: name lengths shouldn't be too short (devoid of meaning) or too long (awkward to read/type) but their length should be "just right". Better to make them too long, because there are ways (such as alt-name) to abbreviate them.

We almost never write the * form of importing. It imports all the names defined in module-name, and "pollutes" our namespace by defining all sorts of names we may or may not use (and even worse, which might conflict and redefine names that we have already defined). Better to explicitly import the names needed/used. Eclipse marks with a warning any names that are imported but unused.

Directly iterating over values in a list vs.
  Using a range to iterate over indexes of values in a list

We know that we can iterate (with a for loop) over ranges: e.g., if alist is a list, we can write

```
alist = [5, 2, 3, 1, 4, 0]
for i in range(len(alist)):          # same as for i in range(0,len(alist)):
    print(alist[i])
```

Here i takes on the values 0, 1, ... , len(alist)-1 but not len(alist), which is 6. The code above prints all six values in alist: alist[0], alist[1], ... alist[5].

Often we want to iterate over all the values in a list (alist) but don't need to know/use their indexes at all: e.g., to print all the values in a list we can use the loop

```
for v in alist:
    print(v)
```

which is much better (simpler/clearer/more efficient) than the loop

```
for i in range(len(alist)):
    print(alist[i])
```

although both produce the same result. Generally, choose to write the simplest loop possible (the one with the fewest details) for all your code. Sometimes you might write a loop correctly, but then realize that you can also write a simpler loop correctly: change your code to the simpler loop. Sometimes (when doing more complicated index processing) we must iterate over indexes. Always try to use the simplest tool needed to get the job done.

In many cases where using range is appropriate, we want to go from the low to high value INCLUSIVELY. I have written function named irange (for Inclusive RANGE) that we can import from the goody module and use like range.

```
from goody import irange
for i in irange(1,10):
    print(i)
```

prints the values 1 through 10 inclusive; range(1,10) would print only 1 through 9. One goal for ICS-33 is to show you how to write alternatives to built-in Python features; we will study how irange is written later in the quarter, but you can import and use it now (and examine its definition in the goody module).

I have also written frange in goody, which allows iteration over floating point (not int) values.

```
from goody import frange
for x in frange(0., 1., .5):
    print(x)
```

```
prints (iterating from 0 to 1 in steps of .5

0.0
0.5
1.0

There are some interesting numerical issues to think about when we use floating
point number:

from goody import frange
for x in frange(0., 1., .2):
    print(x)

which prints

0.0
0.2
0.4
0.6000000000000001
0.8
1.0

Python tries to print floating point values simply, but sometimes it cannot
exactly represent a value, so it prints a value that is a tiny bit lower or
higher. Don't be confused by such numbers.

--------------------------------------------------------------------------------

Arguments and Parameters (and Binding): Terminology (much more details later)

Whenever we DEFINE a function (and define methods in classes), we specify the
names of its PARAMETERS in its header (in parentheses, separated by commas).
Whenever we CALL a function we specify the values of its ARGUMENTS (also in
parentheses, separated by commas). The definition below

def f(x,y):
    return x**y

defines a function of two parameters: x and y. When we define a function we
(re)bind the function's name to the function's object. This is similar to what
happens when we write x = 1 (which (re)binds a name to a data object: an
instance of the int class; if we later write x = 2 we rebind x to a different
object).

Calling  f(5,2*6) calls this function with two arguments: the arguments 5 and
2*6 are evaluated (producing objects) and the values/objects computed from
these arguments (5 and 12) are bound to their matching parameters in the
function header (and then the body of the function is executed). Therefore,
function calls happen in three steps:
  (1) evaluate the arguments
  (2) bind the argument values (objects) to the parameter names
  (3) execute the body of the function, using the values bound to the parameter
      names; so parameters are names inside the name-space of the function,
      along with any local variables the function defines.

We will discuss the details of argument/parameter binding in much more detail
later in these lecture notes, but in a simple example like this one, parameter/
argument binding occurs sequential, binding the first evaluated argument to the
first parameter, the second evaluated argument to the second parameter (etc.).
It is like writing: x = 5 and then y = 2*6 which binds the parameter x to the
object 5 and then the paramter y to the object 12.

Sometimes we can use the parameter of a function as an argument to another
function call inside its body. If we define

def factorial_sum(a,b):
    return factorial(a) + factorial(b)
```

Here the parameters a and b of factorial_sum function are used as arguments in the two calls in its body to the factorial function.

Parameters are always names. Arguments are expressions that evaluate to objects. Very simple expressions include literals (such as 1 and 'abc') and any names bound to values (such as a and b in the calls to factorial above).

It is important that you understand the distinction between the technical term PARAMETER and ARGUMENT, and that calling a function first BINDS that argument values to to their associated parameter names in the functon's HEADER (we will discuss the exact rules later in this lecture note) and then EXECUTES the BODY of the function. Be familar with these related technical terms.

Function calls ... always include ()
  how we can pass a function as an argument to another function

Any time a reference to an object is followed by (...) it means to perform a function call on that object. Some objects will raise an exception if they do not support function calls: writing 3() or 'abc'() will both raise exceptions. Calling 'abc'() raises the TypeError exception, with the description: 'str' object is not callable.

While this rule is simple to understand for the functions that you have been writing and calling, there are some much more interesting ramifications of this simple rule. Run the following code to define these three functions.

```
def double(x):
    return 2*x

def triple(x):
    return 3*x

def times10(x):
    return 10*x
```

Note that each def defines the name of a function and binds that name to the function object that follows it.

If we then wrote

```
f = double
```

then f would become a defined name that is bound to the same (function) object that the name double is bound to (like like writing y = x if x were bound to an integer). The expression "f is double" would evaluate to True, because these two names are bound to the same function object.

Note that there are no () in the code above, so there is no function CALL here: we are just binding names to objects (that happen to be function objects). If we instead wrote f = double(3), then Python would bind f to the int value 6.

Given the assignment f = double above,

```
print( f(5) )
```

Python would print 10, just as it would if we wrote

```
 print( double(5) )
```

because f and double refer to the same function object, and it makes no difference whether we call that function object using the name f or double. The function call does not occur until we use (). Of course the () in print(...) means that the print function is also called: print's argument is the result returned by calling f(5). So the two sets of () in th print(f(5)) means that Python calls two functions while executing print.

Here is a more intersting example, but using exactly the same idea.

```
for f in [double, triple, times10]:
    print( f(5) )
```

Here f is a variable that iterates over (is bound to) all the values in a list (we could also have used a tuple to store these three functions): each value in the list is a reference to a function object (the objects referred to by the names double, triple, and times10). This code in the loop's body prints the values computed by calling each of these function objects with the argument 5. Note that these functions are NOT called when creating the list (no parentheses there!): the list is just built to contain references to these three function objects: again, when their names are not followed by () there is no function call. This code prints 10, 15, and 50.

See the "Creating;/Using a List of Functions" link on the Weekly Schedule to see a picture of how this code works.

Using the same logic, we could also write a dictionary whose keys are strings and whose values are function objects, and then use a string as a key needed to retrieve and call its associated function object.

```
fs = {'x2' : double, 'x3' : triple, 'x10' : times10}
print( fs['x3'](5) )
```

Here fs is a dictionary that stores keys that are strings and associates each with a function object: there are no calls to functions when building the dictionary for fs in the first statement - no (); we then can retrieve the function associated with any key (here the key 'x3') and then call the resulting function object (here fs['x3']) with the argument 5. Of course in the second statement we use the () to call the function selected from the dictionary.

We can also pass (uncalled) functions as arguments to other functions.

```
# p is a predicate: a 1-argument function returning a bool (True or False)
# alist is a list of values on which p can be/is called
# count_true returns the number ov values in alist for which p returns True
def count_true(p, alist):
    count = 0
    for x in alist:
        if p(x):
            count += 1

    return count
```

if we wrote the following (predicate is a module in the ICS-33 course library)

```
from predicate import is_prime
```

(which imports the is_prime function from the predicates module: a PREDICATE is a function of one argument that returns a boolean value) then calling

```
  count_true(is_prime, [2,3,4,5,6,7,8,9,10])
```

returns 4: only the numbers 2, 3, 5, and 7, are prime.

Note that the parameter p is bound to the function object that is_prime is bound to (is_prime is not called when it is specified as an argument), and p is called many times inside the count_true function: one for each value in the list. We call count_true a "functional" because it is passed a function as an argument.

In summary

```
def f(x):
    return 2*x
```

is really just a special syntax for creating a name f and binding it to a new function object (not using an = sign, which we normally using for defining names).

Note the difference between the following, which both print 6. Both g and f refer to the same function object (by the "is" operator, f is g is True). The call g(3) is calling the same function object as the call f(3).

```
x = f(3)
print(x)
```

and

```
g = f
print(g(3))
```

A large part of this course deals with understanding functions better, including but not restricted to function calls: the main thing -but not the only thing- one does with functions; the other main things are passing functions as arguments to other functions (as in count_true) and returning functions from other functions (discussed soon).

So, functions are objects just like integers are objects. Both can have one or more names bound to them. In fact, is some branches of mathematics 3 is considered just the name of a parameterless function: we can call 3() to get its value (although don't try this in Python).

Scope: Global and Local Names (generalized in the next section)

The topic of SCOPE concerns the visibility of variable names: when a name is written in Python code, to what definition of that name does it refer. In this section, we discuss whether naming a variable in a simple context refers to a name defined in the global scopre or local scope; in the next, we generalize this principle in a powerful way, allowing for the creation of functions that return other functions.

Names defined in a module are global definitions; names defined in a function (including its parameters, and later names defined in a class) are local definitions. In a module, we can refer to global names, but not any local names inside the functions/class that are define there. In functions we can refer both to their local names or to global names. Of particular interest is what happens when a name is defined bothglobally and locally.

Here are Python's rules (not all the rules yet, but the ones we focus on now, and correct/expand later), which we will discuss and illustrate below.

1) A name used (to lookup or bind) in a module is global

2) A name used (to lookup or bind) in a function is global if
   a) that name is explicitly declared global (e.g., global x) PRIOR to its use in the function, OR
   b) that name is only looked-up (and never bound) in the function

3) A name used (to lookup or bind) in a function is local, otherwise (e.g., bound in a function AND not declared global before its use)

As a result of these rules, all uses of a name in a given function must be the same: all uses are global or all uses are local.

Let's start by looking at Example 1:

```
x = 1
print(x)
```

which prints 1. Here the module defines a global name x (by rule 1) and binds it to the object 1 and prints the value bound to that global name.

Now let's look at Example 2:

```
x = 1
```

```
def f():
    print(x)

f()
print(x)
```

which prints 1 and then 1. Here the module defines a global name x (by rule 1) and binds it to the object 1; then it defines a global name f (by rule 1) and binds it to a function object: note the function uses the name x, which is global (by rule 2b). When we call f(), it prints the value bound to the global name x. Then, returning to the module, it prints the value bound to the global name x (again, by rule 1).

In this example, all the uses (there is just one) of x inside the function is global.

Now let's look at Example 3:

```
x = 1
def f():
    x = 2
    print(x)

f()
print(x)
```

which prints 2 and then 1. Here the module defines a global name x (by rule 1) and binds it to the object 1; then it defines a global name f (by rule 1) and binds it to a function object: note the function uses the name x twice, which is local (by rule 3: it is bound in the function and is not declared global in the functioni). When we call f(), it binds the local name x to 2 and then prints the value bound to the local name x. Then, returning to the module, it prints the value bound to the global name x (by rule 1), which is still 1.

What is the primary difference between this example and the preceding one? Here, both uses of the name x in this function are local (not global).

Now let's look at Example 4:

```
x = 1
def f():
    y = 2
    print(x,y)

f()
print(x)
```

which prints 1 2 and then 1. Here the module defines a global name x (by rule 1) and binds it to the object 1; then it defines a global name f (by rule 1) and binds it to a function object: note the function uses the names x and y; x is global (by rule 2b) and y is local (by rule 3). When we call f(), it binds the local name y to 2 and then prints the values bound to the global name x and the local name y. Then, returning to the module, it prints the value bound to the global name x (by rule 1).

Note that if we replaced print(x) at the end of the module with print(x,y) Python would raise a NameError exception because there is no global name y; y is a local name defined only in the function f.

Something interesting happens by the rules above, if we try to get function f to not only print the original value of the global name x, but also try to rebind that global name to 2. Let's see why this fails to be the effect of the code below (later we will see how to accomplish this task). In fact, it fails in an interesting way.

Now let's look at Example 5:

```
x = 1                   # wrong code to print a global and then change it
def f():                # wrong code to print a global and then change it
```

```
    y = 2               # wrong code to print a global and then change it
    print(x,y)          # wrong code to print a global and then change it
    x = 2               # wrong code to print a global and then change it
                        # wrong coed to print a global and then change it
f()                     # wrong code to print a global and then change it
print(x)                # wrong code to print a global and then change it
```

Here, the module defines a global name x (by rule 1) and binds it to the object 1; then it defines a global name f (by rule 1) and binds it to a function object: note the function uses the names x and y; x is local (by rule 3; it is bound in the function and not declared global) and y is local (also by rule 3, ditto). When we call f(), it binds the local name y to 2 and then tries to print the values bound to the local names x and y; but here there is no binding (yet) for x -even though Python knows that x must be local- so Python raises the UnboundLocalError exception: local variable 'x' referenced before assignment.

So, it does NOT look-up x as a global in the print and then define x as a local! The name x is either always global or always local in a function.

If we put the x = 2 before the print, it doesn't raise an exception, but doesn't rebind the global name x either.

Now let's look at Example 6:

```
x = 1                   # wrong code
def f():                # wrong code
    y = 2               # wrong code
    x = 2               # wrong code
    print(x,y)          # wrong code
                        # wrong code
f()                     # wrong code
print(x)                # wrong code
```

Python would print 2 2 and then 1. Here, the module defines a global name x (by rule 1) and binds it to the object 1; then it defines a global name f (by rule 1) and binds it to a function object: note the function uses the names x and y; x is local (by rule 3; it is bound in the function and not declared global) and y is local (also by rule 3, ditto). When we call f(), it binds the local name y to 2 and binds the local name x to 2. Then it print the values bound to the local names x and y. Then, returning to the module, it prints the value bound to the global name x (which has remained bound to 1).

We can change the value in the global name x by using a "global" declaration before x is used inside f. This declaration means that x should always be treated as a global name inside this function (whether it is examined or bound to an object, or both) by rule 2.

Now let's look at Example 7:

```
x = 1
def f():
    global x
    y = 2
    print(x,y)
    x = 2

f()
print(x)
```

Python would print 1 2 and then print 2 (thus changing the global name x).

Here, the module defines a global name x (by rule 1) and binds it to the object 1; then it defines a global name f (by rule 1) and binds it to a function object: note the function uses the names x and y; x is global (by rule 2a; it is declared global) and y is local (by rule 3, it is bound in the function but not declared global). When we call f(), it binds the local name y to 2 and then prints the values bound to the global name x and the local name y. Then it

rebinds the global name x to 2. Finally, returning to the module, it prints the value bound to the global name x (which has now been rebound to 2).

So, if a function must (re)bind a global name, the functioin must explicitly declare that name global.

Note that we we wrote "global x" after print(x,y) in f, this x would be considered a local name (rule 2a does not apply), so Python would instead raise a SyntaxError: name 'x' is used prior to global declaration.

To repeat, when Python creates an function object, it first scans the whole function to determine whether a name refers to a global name or a local name, looking for global declarations and bindings to make this determination, using the rules stated above. The rule for 2a hinges on the word PRIOR. So if is declared global subsequent to its first use, it is considered to be in error.

In summary, we can always lookup (find the value of, evaluate) a global name inside a function to find its value without doing anything special, but if we want to (re)bind the global name to a new value inside the function, we must declare the name global somewhere in the function (before its first use). If we forget to declare the name global and (re)bind the name, Python will instead define it as a local name (rule 3) and all occurrences of that name in the function will use it as a local name (so it better be defined first).

What do you think the following code will do? Notice that is similar to the code above, but it does not define a global name x before defining f. Can you explain why (in terms of the rules) it does what it does?

Now let's look at Example 8:

```
def f():
    global x
    x = 2
    y = 2
    print(x,y)
f()
print(x)
```

Also can you explain what would happen if the print statement print(x,y) were moved between the statements y = 2 and x = 2, ore moved before both?

Bottom Line: You should understand how Python decides whether names in a function are global or local. TYPICALLY, no global names should be used: all information that a function uses should be passed into the function and bound to its parameters. BUT if a function must (re)bind a global name either

(1) the name should be declared global (so it can be examined/changed), or

(2) the function should be called like global-name = f(...) so that the global name is not declared/changed as a global inside the function, but is instead changed as a result of an assignment statement made after calling the function. We can even use this mechanism for rebinding multiple global names: gn1, gn2, ..., gnN = f(...) where f returns an N-tuple or N-list (see the Parallel/Tuple/List Assignment -aka sequence unpacking- section below).

Problem 1: Explain why we cannot write a general function in Python that exchanges the references in its argument names. That is, we cannot write

```
def f(x,y):
  ...
such that
a = 1
b = 2
f(a,b)
print(a,b) # prints 2 and 1
```

Such a function canno be written in Java, but can be written in C++.

Problem 2: Write a SPECIFIC function in Python that exchanges the references

ONLY FOR THE NAMES a AND b. That is,

```
def f():
  ...
such that
a = 1
b = 2
f()
print(a,b) # prints 2, and 1
```

Problem 3: Write a GENERAL function in Python that we can CALL IN A SPECIAL WAY to exchanges the references of its two argument names. Hint: read the paragraph above.

```
def f(x,y):
  ...
a = 1
b = 2
# some Python statement, calling f(a,b) such that
print(a,b) # prints 2, and 1
```

These local/global rules are generalized in the next section to include 4 different locations/scopes: LEGB

1) functions (Local scope)
2) enclosing functions (Enclosing scope)
3) modules (Global scope)
4) the special builtins module (Builtins scope)

Generalizing Scope: Local, Enclosing, Global, and Builtins
(applicaton: a function call that return a reference to a function object)

Look at the following functions, which all define a local function (and define local variables too), and then return the value of that local function. This feature in Python (and the feature of passing function objects as arguments to parameters; see the count_true function defined earlier) are very powerful and useful. In this section, we will generalize the scope rules discused above and learn how these new rules apply when functions defined in functions are called).

```
def bigger_than(age):
    def test_it(x):
        return x > age

    return test_it

def running_average1():
    data = []              # Mutated, but not rebound, in include

    def include_it(x):
        data.append(x)     #   mutate data's list
        return sum(data)/len(data)

    return include_it

def running_average2():
    sum   = 0              # Mutated, but not rebound, in include
    count = 0              # Mutated, but not rebound, in include

    def include_it(x):
        nonlocal sum,count
        sum += x           #   rebind sum
        count+= 1          #   rebind count
        return sum/count

    return include_it
```

Note that the body of these functions, and the bodies of their inner functions

(test_it and include_it) can also use global names (as discused in the last section); but none of these do, which follows good practice for defining any functions.

We will now write the complete rules Python uses to look up and bind names. Thes rules generalizes what we learned in the previous section: we will now define any name in a function as either local, nonlocal/encloseing, global, or LEGB; then we will learn how such names are looked-up and bound. Here are the rules from the previous sections, restated and extended using these new terms. We separate rules into how names are classified and how they are looked-up/bound.

1) A name used in a module is global

2) A name used in a function is global if that name is explicitly declared global (e.g., global x) PRIOR to its use in the function

3) A name used in a function is nonlocal/enclosing if that name is explicitly declared nonlocal (e.g., nonlocal x) prior to its use in that function

4) A name used in a function is local
   a) that name is a parameter in that function, or
   b) that name is bound anywhere in that function (and is neither declared global nor nonlocal)

5) Otherwise, a name is LEGB (if it is none of the above)
   NOTE: an LEGB name can NEVER be bound in a function (if it is bound, it must be declared global, or nonlocal, or will be local by 4a or 4b).
   So LEGB names are ONLY LOOKED-UP.

If we import a module name (call it M), then we can lookup/rebind any global name defined in that module (call it n) by writing M.n: e.g., print(M.n) or M.n = ...

Here are the corresponding rules for looking-up/binding all these names.

A+B) When a name is global,
   1) Python LOOKS UP that name, in order,
      a) in the Global scope; and if not found
      b) in the Builtins scope (in the builtins module); and if not found
      c) raises NameError
   2) Python BINDS that name in the Global scope

C) When a name is nonlocal/enclosing, Python looks for the name in the scope of the function enclosing it (if there is one); if it doesn't find it there, it looks in the scope of the function enclosing the enclosing function (if there is one). This process continues outward until it finds the name and then uses the name in that scope. If it cannot find the name and reaches the global scope, Python raises a SyntaxError exception: no binding for nonlocal ... found

D) When a name is local, Python looks-up/binds that name in the local scope.

E) When a name is LEGB, Python looks for the name, in order,
   1) in the   Local scope of the function; and if not found
   2) in any of the Enclosing scopes (see rules in C); and if not found
   3) in the   Global scope; and if not found
   4) in the   Builtins scope(in the builtins module); and if not found
   5) raises NameError

For example, the code

```
x = 1
def f(y):
    print(x,y)

f(2)
print(x)
```

prints 1 2 and then 1. Here the module defines a global name x (by rule 1) and binds it to the object 1; then it defines a global name f (by rule 1) and binds it to a function object: note the function uses the names print, x and y (yes, now we even look at where the name print comes from); print is LEGB (by rule 5), x is LEGB (by rule 5) and y is local (by rule 4a). After the function definition the name f is global (by rule 1) and the the name print is LEGB (by rule 5) and the name x is global (by rule 1).

When we call f(2),

Python looks up the global name f and finds it in the global scope where it is declared and calls that function, first binding its local name y to the object 2; then it executes the body of the function. It looks up the object associated with the LEGB name print (which it finds not in the local scope, not in an enclosing scope -there is none-, not in the global scope, but finally in the builtins scope): the print function is defined in the builtins module. It then calls the print function after looking up the binding of the LEGB name x (which it finds not in the local scope, not in an enclosing scope -there is none-, but finally in the global scope, with a binding 1) and the local name y (it finds its binding 2, in the local scope). After printing 1 2 the function terminates.

Now, Python looks up the object associated with the global name print (which it finds not in the global scope, but in the builtins scope). It then looks up the global name x (which it finds in the global scope, with a binding of 1) and calls the print function, printing the value 1. The module then terminates.

******
This is an incredibly detailed explanation of what happens, but it is an accurate description. We will do many more examples in class, but it is up to you to study these and be able to know/describe/hand-simulate exactly what happens.
******

-----Back to understand a function defined in a function

Now let's return to the three concrete examples of a function defined inside a function. None of these functions use global or builtin names, so all names are defined locally or nonlocally (in an enclosing scope, found by the LEGB rule).

The first is function is

```
def bigger_than(age):
    def test_it(x):
        return x > age

    return test_it
```

Note that inside the function bigger_than, age is a local name (this name is used only inside test_it). Inside the function test_it, x is a local name and age is an LEGB name (by rule 5); this is the only place age is used.

Lets look at the following sequence of statements

```
old = bigger_than(60)
print(old(65))
```

Python executes the first statement by calling bigger_than and then binding the object that it returns to the variable name old. When calling bigger_than, Python first binds the argument 60 to the parameter age, and then executes its body, whose two statements

  (1) define test_it to refer to a NEW function object (new each time bigger_than is called). That function object, being enclosed in bigger_than, stores a CLOSURE consisting of all the local variables and their bindings in bigger_than: here just age bound to 60. A function stores a closure only if it is defined inside an enclosing function, so global functions store no closures. We can think of a closure as just a dictionary storing variable names and their current bindings.

(2) return a reference to the function object bound to the local variable test_it. In the statement above, the global name old is bound to this returned function object.

When old(65) is called, Python first binds the argument 65 to the parameter x, and then executes old's body. It is just one return statement, which compares the local name x to the LEGB name age; it looks-up the value for age inside the closure of old's function object (where age is bound to 60). Since 65>60 evaluates to True, old(65) returns True, which is printed. (*of course, the binding of the parameter x inside this function call is 65).

See a link to the picture of a Function returning a function in the lectures and/or weekly schedule on the course website.

If we wrote the following

```
old     = bigger_than(60)
ancient = bigger_than(90)
print (old(10), old(70), old(90), ancient(70), ancient(95))
```

Python prints: False True True False True

Each assignment statement binds its name (old and ancient) to a different function object that is created/returned by different calls to the bigger_than function. Each of these function objects has its own closure: age is bound to 60 in old's function object's closure; age s bound to 90 in ancients's function object's closure. So the LEGB name age in each different function will get its value from a different closure, so the functions can return different results.

In fact, we even could have avoided the local variable old and written

```
print ( bigger_than(60)(70) )
```

We know that bigger_than(60) calls bigger_than with the argument 60, which returns a result that is a reference to its inner function test_it in whose closure age is bound to 60; so by writing (70) after that, we are just calling tht returned function object (the one bigger_than(60) returned). When calling this returned function object using the argument 70, it returns True.

In the function

```
def running_average1():
    data = []              # Mutated, but not rebound, in include

    def include_it(x):
        data.append(x)     #   mutate data's list
        return sum(data)/len(data)

    return include_it
```

the include_it function refers to data, which is an LEGB name, multiple times. Note that it is an LEGB name in include_it, because it is not bound in that function. So, each time running_average1 is called, it creates and returns a new function object bound to include_it, whose closure will contain the name data bound to an empty list. Each time we call it, it will mutate data by appending another value to it, and returning the average of all the values it has been called with (now in the list). For example

```
mileage_per_tank = running_average1()
print(mileage_per_tank(300))
print(mileage_per_tank(200))
print(mileage_per_tank(400))
```

prints 300 (300/1), 250((300+200)/2), and 300 ((300+200+400)/3)

So, using functions returning functions, a function can remember information from its previous calls! It stores such information in the closure for its enclosing function (why the name "closure" is meaningful).

Finally, in the function

```
def running_average2():
    sum   = 0              # Mutated, but not rebound, in include
    count = 0              # Mutated, but not rebound, in include

    def include_it(x):
        nonlocal sum,count
        sum += x           #   rebind sum
        count+= 1          #   rebind count
        return sum/count

    return include
```

the include_it function refers to sum and count, which because they are BOUND inside include_it (using +=) would normally be local names. But as local names they would not be initialized and cause an error. Instead, we want the include_it function to use the sum/count initialized in running_average2, so we explicitly declare them nonlocal, so their values will be LOOKED-UP and REBOUND in the closure for the running_averqage2 function call in which the include_it function was created.

So, each time running_average2 is called, it creates and returns a new function object bound to include_it, whose closure will contain the names sum and count bound to 0. Each time we call it, it will rebind sum and count by incrementing them, and returning the average of all the values it has been called with. Thi function has the same behavior as running_average1 by storing just two integers, without having to remember a list of values. But doing so requires nonlocal variables.

The bottom line here is that there are complicated rules that govern the scope of variable names. But they are often used in stardard ways, to get powerful behavior from simple Python code.

Problem: Predict what the following prints. Drawing a diagram might help.

```
def f():
    prev = None

    def g1(x):
        nonlocal prev
        temp = prev
        prev = x+5
        return temp
    def g2(x):
        nonlocal prev
        temp = prev
        prev = 5*x
        return temp
    return g1,g2

f1,f2 = f()
f3,f4 = f()
print(f1(1))
print(f2(2))
print(f3(5))
print(f4(6))
```

Problem: What would the difference be if we defined a global vaiable prev = 100 and change the nonlocal declaration in g2 to be a global declaration?

Functions vs Methods: The Fundamental Equation of Object-Oriented Programming

Functions are typically called f(...) while methods are called on objects like o.m(...). Think of x ='candide' followed by calling print(x.replace('d','p'))

In reality, a method call is just a special syntax to write a function call. The special "argument" o (normally arguments are written inside the parentheses) prefixes the method name. Functions and methods are related by what I call "The Fundamental Equation of Object-Oriented Programming."

```
  o.m(...)     =     type(o).m(o,...)
```

On the right side
1) type(o) returns a reference to the class object o was constructed from.
2) .m means call the function m declared inside that class: look for def m(self,...): .... in the class
3) pass o as the first argument to m: recall, that when defining methods in classes we write def m(self, ...); where does the argument matching the self parameter come from? It comes from the object o in calls like o.m(...)

So, calling 'candide'.replace('d','p') is exactly the same as calling str.replace('candide','d','p'), because type('candide') returns a reference to the str class, which defines many string methods, including the replace method.

How well do you understand self (or your-self, for that matter:)? This equation is the key. I believe a deep understanding of this equation is the key to clarifying many aspects of object-oriented programming in Python (whose objects are constructed from classes). Just my two cents. But we will often return to this equation throughout this course. I've never seen any books that talk about this equation explicitly. We will revisit FEOOP when we spend a week discussing how to write more sophisticated classes than the ones you wrote in ICS-32.

Oh, by the way, I must say that this equation is true, but not completely true. As we will later see: (a) if m is in the object o's namespaces, it will be called directly (bypassing looking in o's class/type), and (b) when we learn about class inheritance, the inheritance hierarchy of a class provides other classes in which to look for m if it is not declared directly in o's class/type. So, as in courtroom testimony, we need to distinguish the truth, the whole truth, and nothing but the truth.

But this equation is so simple, clear (once you understand it), and useful for tons of examples, it is worth memorizing, even if it is not completely accurate (yet).

---

lambda:

Lambdas are used in expressions where we need a very simple function. A lambda represents a special function object. Instead of defining a full function (with a def and a name for it), we can just use a lambda: after the word lambda comes its parameters separated by commas, then a colon followed by a single EXPRESSION that computes the value of a lambda (no "return" is needed, and the function cannot include control structures/statments, not even a sequence of statements).

So, writing ...(lambda x,y : x+y)... in some context

Is just like first defining

```
def f(x,y):
    return x+y
```

and then writing ...f... in the same context. That is calling print((lambda x,y : x+y)(2,3)) is the same as calling print(f(2,3)).

A lambda produces an UNNAMED function object. For example, we can also write the following code, whose first line binds to the name f the unnamed lambda/function object, and whose second line calls the unnamed lambda object via the name.

```
g = lambda x,y : x+y  # lambdas have one expression after : without a return
print( g(1,3) )
```

and Python will print 4. I often put lambdas in parentheses, to more clearly

denote the start and end of the lambda, for example writing

```
g =  (lambda x,y : x+y)
```

Using this form, we can write code code above without defining g, writing just

```
print( (lambda x,y : x+y)(1,3) )
```

In my prompt module (MY PREFERRED WAY OF DOING PYTHON USER-INPUT), there is a function called for_int that has a parameter that specifies a boolean function (a function taking on argument and returning a boolean value: we will repeated use the word predicate for this kind of function) that the int value must satisfy, to be returned by the prompt (otherwise the for_int function prompts the user again, for a legal value).

That is, we pass a function object (without calling it) to prompt.for_int, which CALLS that function on the value the user enters to the prompt, to verify that the function returns True for that value.

So the following code fragment is guaranteed to store a value between 0 to 5 inclusive in the variable x. If the user enters a value like 7, an error will be printed and the user reprompted for the information.

```
import prompt
x = prompt.for_int('Enter a value in [0,5]', is_legal = (lambda x : 0<=x<=5))
print(x)
```

Execute this code in Python.

In a later part of this lecture note, we will see how to use lambdas to simplify sorting lists (or simplify iterating through any data structures according to an ordering function).

Again, I often put lambdas in parentheses for clarity: see the prompt.for_int example above. But there are certain places where lambdas are required to be in parentheses: assume def g(a,b): ... and the lambda below will receive a 2-tuple as an argument and return the reversed 2-tuple. Calling
g(1, (lambda x : x[1],x[0])) requires the lambda to be in parentheses, because without the parentheses it would read as g(1, lambda x : x[1],x[0]). In this function call, Python would think that x[0] was a third argument (not part of the lambda) to function g, which would raise an exception when called, because g defined only two parameters. Of course, we could also use different parentheses in the call and write g(1, lambda x : (x[1],x[0])). We also wrote this call above as g(1, (lambda x : x[1],x[0])). In both cases the parentheses remove the ambiguity.

In a previous section (functions returning functions) we wrote the code

```
def bigger_than(age):
    def test_it(x) :
        return x > age
    return test
```

We defined the test_it function just so we could return a reference to it. Now, because the test_it function is so simple, and the only place we refer to it by name is in the return statement, we can simplify this code by returning a lambda instead of defining/returning a named function:

```
def bigger_than(age) :
    return (lambda x : x > age)
```

So, in each version of the bigger_than function, it returns a reference to a function object.

Parallel/Tuple/List Assignment (aka sequence unpacking)

Note that we can write code like the following: here both x,y and 1,2 are

implicit tuples; and we can unpack them as follows

```
x,y = 1,2
```

In fact, you can replace the left hand side of the equal sign by either (x,y) or [x,y] and replace the right hand side of the equal sign by either (1,2) or [1,2] and x gets assigned 1 and y get assigned 2: even (x,y) = [1,2] works correctly.

In most programming languages (including Java and C++), to exchange the values of two variables x and y we write three assignments (can you prove that writing just x = y followed by y = x fails to exchange these values?):

```
temp = x
x    = y
y    = temp
```

Problem: why would x = y followed by y = x not do the swapping?

In Python we can write this code using one tuple assignment

```
x,y = y,x
```

To do any parallel assignment, Python
  (a) first computes all the values of the expressions/objects on the right
        (1 and 2 from the top)
  (b) second binds each name on the left (x then y) to these computed
        values/objects (binds x to 2, then bind y to 1)

This is similar to how Python calls functions: first evaluating all the arguments and then binding them to all the parameters.

This is also called "sequence unpacking assignment": both tuples and list are considered kinds of sequences (where order is important) in Python. Note that x,y = 1,2,3 and x,y,z = 1,2 would both raise ValueError exceptions, because there are different numbers of names and values to bind to them. We will frequently use simple forms of parallel/unpacking assignment when looping through items in dictionaries (illustated below; used extensively later in this lecture note), but even more complicated forms are possible: for example.

```
l,m,(n,o) = (1, 2, [3,4])
print(l,m,n,o)
```

prints: 1 2 3 4

Here, each name is bound to one int value. Python also allows writing

```
a,*b,c = [1,2,3,4,5]
print(a,b,c)
```

which prints as: 1 [2, 3, 4] 5

Here, * can preface any name; the name is bound to a sequence of any number of values, so as to correctly bind a and c. That is, *b would need to bind to a sequence of all but the first and last value. We could not write

```
*a,*b,c = [1,2,3,4,5]
```

because there is no unique way to bind a, b, and c: for example a=1, b=[2,3,4], and c=5 works; but so does a=[1,2],b=[3,4],c=5.

In fact, we can write complicated parallel assignments with multiple *s like

```
l,(*m,n),*o  = (1, ['a','b','c'], 2, 3,4)
print(l,m,n,o)
```

so long as there is a unique way to decide what the * variables bind to.

The above statement prints as: 1 ['a', 'b'] c [2, 3, 4]

Generally sequence unpacking assignment is useful if we have a complex tuple/list structure and we want to bind names to its components, to refer to these components more easily by these names: each name binds to part of the complicated structure.

As another example, if we define a function that returns a tuple

```
def reverse(a,b) :
    return (b,a)    # we could also write just return b,a
```

we can also write x,y = reverse(x,y) to also exchange these values.

Finally, one of the most common ways to use unpacking assignment is in for loops: for example, we can write the following for loop to print the sum of each triple in the list

```
for i,j,k in [(1,2,3), (4,5,6), (7,8,9)]:
    print (i+j+k)
```

this is much simpler and clearer than using one name for the entire tuple, and then indexing that one name.

```
for t in [(1,2,3), (4,5,6), (7,8,9)]:
    print (t[0] + t[1] + t[2])
```

The loop above assigns i, j, and k the three values in each 3-tuple in the list that the for loop is iterating over. We will see more about such assignments when iterating though items in dictionaries. A preview is

```
for k,v in d.items():  #d is any dictionary
    print(k,'->',v)    #  print its key and value pairs (abbreviated k,v)
```

which prints each key and its associated value for each item (each key/value association) that we are iterating through in a dictionary.

```
d = {'a':1, 'b':2, 'c':3, 'd':4}
for k,v in d.items():
    print(k,'->',v)
```

prints

```
d -> 4
a -> 1
c -> 3
b -> 2
```

although the keys/values may print in any order

This is simpler than the following loop, which iterates over the tuples produced when iterating over d.items().

```
for t in d.items():         #d is any dictionary
    print(t[0],'->',t[1])   #  print its key (t[0])and value (t[1]) pairs
```

When you write for loops that loop over complicated data (as in dictionary items), use multiple variables to name the indvidual parts of the data. Multiple names for the parts often make the loop body's code easier to write and understand. Students often fail to follow this advice and eventually get docked points for writing code that is more complicated than needed.

End of 1st Lecture on this material

Iterable

When we specify that an argument of a function must be iterable, it might be one of the standard data structures in Python: str, list, tuple, set, dict. All these data structures are iterable: we can iterate over the values they contain by using a simple for loop.

But we will learn other Python features (classes and generators) that are also iterable. The difference is, that for standard Python data structures we can call the "len" function on them and index them [...]; but for general iterable arguments we CANNOT call "len" nor index them: we can only iterate over their values with a for loop: getting the first value, the second value, etc. (never knowing when there won't be a next value). Later in the quarter we will learn how to call iter and next explicitly on iterables; for loops implicitly call these two functions on iterables.

Also, we can always use a comprehension (dicussed later in this lecture) to transform any iterable into a list or tuple of its values, doing so takes up extra time and space, and should be avoided unless necessary. But learning how to convert an iterable into a list or tuple is instructive.

---

sort (a list method)/sorted (a function): and their "key"/"reverse" parameters

First, we will examine the following similar definitions of sort/sorted. Then we will learn the differences between these similar and related features below.

(1) sort is a method defined on arguments that are LIST objects; it returns None but MUTATES its list argument to be in some specified order: e.g., alist.sort() or alist.sort(reverse=True); note that calling print(alist.sort()) sorts (mutates) alist and prints None (sort's returned value).

(2) sorted is a function defined on arguments that are any ITERABLE object; it returns a LIST of its argument's values, in some specified order. The argument itself is NOT MUTATED: e.g., sorted(adict) or sorted(adict,reverse=True). Neither changes adict to be sorted, because dictionaries are never sorted; both produce a LIST of sorted keys in adict. We can also call sorted with an argument that is a str, list, tuple, set, ... anything that is iterable in Python.

So, the sort method can be applied only to lists, but the sorted function can be applied to any iterable (str, list, tuple, set, dict, etc.)

For example, if votes is the list of 2-tuples (candidates, votes) below, we can execute the following code.

```
votes = [('Charlie', 20), ('Able', 10), ('Baker', 20), ('Dog', 15)]
votes.sort()
for c,v in votes:                          # note parallel/unpacking assignment
    print('Candidate', c, 'received', v, 'votes')
print(votes)
```

The call votes.sort() uses the sort METHOD to sort the LIST (MUTATE it); then the for loop iterates over this newly-ordered list and prints the information in the list in the order it now appears. When the entire votes list is printed after the loop, we see the list has been MUTATED and is now sorted. It prints

```
Candidate Able received 10 votes
Candidate Baker received 20 votes
Candidate Charlie received 20 votes
Candidate Dog received 15 votes
[('Able', 10), ('Baker', 20), ('Charlie', 20), ('Dog', 15)]
```

We will see why it sorts by name soon, and learn how to use the key argument to sort by anything else (say by votes).
Contrast this with the following code.

```
votes = [('Charlie', 20), ('Able', 10), ('Baker' ,20), ('Dog', 15)]
for c,v in sorted(votes):                # note parallel/unpacking assignment
    print('Candidate', c, 'received', v, 'votes')
print(votes)
```

Here we never sort/mutate the votes list. Instead we use the sorted FUNCTION, which takes an ITERABLE argument (a LIST is iterable) and returns a NEW LIST that is sorted (leaving votes unchanged). Then we iterate over that returned list to print its information (which is a sorted version of votes). But here, when we print the votes list at the end, we see that the votes list remained unchanged. It prints

```
Candidate Able received 10 votes
Candidate Baker received 20 votes
Candidate Charlie received 20 votes
Candidate Dog received 15 votes
[('Charlie', 20), ('Able', 10), ('Baker', 20), ('Dog', 15)]
```

The sorted function can be thought of as first iterating over the parameter, creating a temporary list; then sorting that temprorary list; and finally returning the sorted temporary list: sorted mutates the list it creates internally; it does not mutate the argument matching iterable.

So, we can think of the sorted function to be defined by

```
def sorted (iterable):
  alist = list(iterable)  # create a list with all the iterable's values
  alist.sort()            # sort that list using the sort method for lists
  return alist            # return the sorted list
```

Question: What would the following code do? If you understand the definitions above, you won't be fooled and the answer won't suprise you. Hint: what does votes.sort() return vs. sorted(votes)?

```
votes = [('Charlie', 20), ('Able', 10), ('Baker' ,20), ('Dog', 15)]
for c,v in votes.sort():
    print('Candidate', c, 'received', v, 'votes')
print(votes)
```

If we were going to print some list in a sorted form many times, it would be more efficient to sort/mutate it once, and then use a regular for loop on that sorted list (either mutate the original or create a new, sorted list and use it multiple time). But if we needed to keep the list in a certain order and care about space efficiency and/or don't care as much about time efficiency, we would not sort/mutate the list and instead call the sorted function whenever we need to process the list in a sorted order.

Note that if we store votes_dict as a dict data structure (a dictionary associating candidates with the number of votes they received), and then tried to call the sort method on it, Python would raise an exception

```
votes_dict = {'Charlie': 20, 'Able': 10, 'Baker': 20, 'Dog': 15}
votes_dict.sort()
```

```
  AttributeError: 'dict' object has no attribute 'sort'
```

The problem is: the sort method is defined only on LIST class objects, not DICT class objects.

So, Python cannot execute votes_dict.sort()! It makes no sense to sort a dictionary. In fact, we cannot sort strings (they are immutable); we cannot sort tuples (their order is immutable); we cannot sort sets (they have no order, which actually allows them to operate more efficiently; we'll learn why later in the quarter); we cannot sort dicts (like sets, they have no order, which allows them to operate more efficiently; ditto).

BUT, WE CAN CALL THE SORTED FUNCTION ON ALL FOUR OF THESE DATA STRUCTURES, and generally on anything that is iterable: As we saw, Python executes the sorted

function by creating a temporary list from all the values produced by the iterable, then sorts that temporary list, and then returns the sorted list. Here is one example of how the sorted function processes votes_dict. Note that executing sorted(votes_dict) is the same as executing sorted(votes_dict.keys()) which produces and iterates over a sorted list of the dict's keys.

```
votes_dict = {'Charlie' : 20, 'Able' :  10, 'Baker' : 20, 'Dog' : 15}
for c in sorted(votes_dict):       # same as: for c in sorted(votes_dict.keys())
    print('Candidate', c, 'received', votes_dict[c], 'votes')
print(votes_dict)
```

It prints

It would show as
Candidate Able received 10 votes
Candidate Baker received 20 votes
Candidate Charlie received 20 votes
Candidate Dog received 15 votes
{'Charlie': 20, 'Able': 10, 'Baker': 20, 'Dog': 15}

-----Start alternative example
Note: if we wrote

```
votes_dict = {'Charlie' : 20,  'Able' : 10,  'Baker' : 20,  'Dog' : 15}
print(sorted(votes_dict))
```

Python prints a list, returned by sorted, of the dictionary's keys in sorted order:

['Able', 'Baker', 'Charlie', 'Dog']
-----Stop alternative example

Well, this is one "normal" way to iterate over a sorted list built from a dictionary. We can also iterate over sorted "items" in a dictionary as follows (the difference is in the for loop and the print). We will examine more about dicts and the different ways to iterate over them later in this lecture. Recall that each item in a dictionary is a 2-tuple consisting of one key and its associated value.

```
votes_dict = {'Charlie' : 20,  'Able' : 10,  'Baker' : 20,  'Dog' : 15}
for c,v in sorted(votes_dict.items()):     # note parallel/unpacking assignment
    print('Candidate', c, 'received', v, 'votes')
print(votes_dict)
```

It prints identically to the original example using dicts.

Notice that this print doesn't access votes[c] to get the votes: that is the second item in each 2-tuple being iterated over using .items(). This is because iterating over votes_dict.items() produces a sequence of 2-tuples, each containing one key and its associated value. The order that these 2-tuples appear in the list is unspecified, but using the sorted function ensures that the keys appear in order.

-----Start alternative example
Note: if we wrote

```
votes_dict = {'Charlie' : 20,  'Able' : 10,  'Baker' : 20,  'Dog' : 15}
print(list(votes_dict.items()))
print(sorted(votes_dict.items()))
```

Python first prints a list of the dictionary's items in an unspecified order, then it prints a list of the same dictionary's items, in sorted order:

```
[('Charlie', 20), ('Able', 10), ('Baker', 20), ('Dog', 15)]
[('Able', 10), ('Baker', 20), ('Charlie', 20), ('Dog', 15)]
```

both list(...) and sorted(...) produce a list of the items in the dictionary, with only sorted(....) producing a list that is guaranteed to be sorted.
-----Stop alternative example

How to use a key parameter to sort arbitrarily:

How does sort/sorted work? How do they know how to compare the values they are sorting: where should come before which? There is a standard way to compare any data structures, but we can also use the "key" and "reverse" parameters (which must be used with their names, not positionally) to tell Python how to do the sorting. The reverse parameter is simpler, so let's look at it first; writing sorted(votes,reverse=True) sorts, but in the reverse order (writing reverse=False is like not specifying reverse at all). So, returning to votes as a list,

```
votes = [('Charlie', 20), ('Able', 10), ('Baker' ,20), ('Dog', 15)]
print(sorted(votes,reverse=True))
```

prints the list of 2-tuples

```
[('Dog', 15), ('Charlie', 20), ('Baker', 20), ('Able', 10)]
```

What sort is doing is comparing each value in the list to the others using the standard meaning of <. You should know how Python compares str values, int values, etc. But how does Python compare whether one tuple is less than another? Or whether one list is less than another? The algorithm in analagous to strings, so let's first re-examine comparing strings. The ordering, by the way, is called "lexicographic ordering", and also "dictionary ordering" (related to the order words appear in dictionaries that are books, not Python dicts/dictionaries).

Interlude: Comparing Strings in Python:

String comparison: x to y (high level: how does x compare to y1):

Find the first/minimum index i such that the ith character in x and y are different (e.g., x[i] != y[i]). If that character in x is less than that character in y (by the standard ASCII table) then x is less than y; if that character in x is greater than that character in y then x is greater than y; if there is no such different character, then how x compares to y is the same as how x's length compares to y's length (either less than, equal or greater than).

Here are three examples:

(1) How do we compare x = 'aunt' and y = 'anteater'? The first/minimum i such that the characters are different is 1: x[1] is 'u' and y[1] is 'n'; 'u' is greater than 'n', so x > y, so 'aunt' > 'anteater'.

(2) How do we compare x = 'ant' and y = 'anteater'? There is no first/minimum i such that the characters are different; len(x) < len(y), so x < y, so 'ant' < 'anteater'. The word 'ant' appears in an English dictionary before the word 'anteater' ('ant' is a prefix of 'anteater').

(3) How do we compare x = 'ant' and y = 'ant'? There is no first/minimum i such that the characters are different; len(x) == len(y), so x == y, so 'ant' == 'ant'. I show this example, which is trivial, just to be complete.

See the Handouts(General) webpage showing the ASCII character set, because there are some suprises. You should memorize that the digits and lower/upper case letters compare in order, and all digts < all upper-case letters < all lower-case letters. So 'TINY' < 'huge' is True because the index 0 characters different, and 'T' is < 'h' (all upper-case letters have smaller ASCII values than any lower-case letters). Likewise, 'Aunt' < 'aunt' because 'A' < 'a'. Note that we can always use the upper() method (as in x.upper() < y.upper()) or lower() method to perform a comparison that ignores the case of the letters in a string: 'TINY'.lower() > 'huge'.lower() because 'tiny' > 'huge' because 't' > 'h'.

Use these rules to determine whether '5' < '15' by comparing the string '5' to '15'. What is your answer; what is the correct answer?

Back to comparing tuples (or lists). We basically do the same thing. We look at what is in index 0 in the tuples; if different, then the tuples compare in the same way as the values in index 0 compare; if the values at this index are equal, we keep going until we find a difference, and compare the tuples the same way that the differences compare; if there are no differences, the tuples compare the same way their lengths compare.

So ('UCI', 100) < ('UCSD', 50) is True because at index 0 of these tuples, the string values are different; and 'UCI' < 'UCSD' (because at index 2 of these strings, the character values are different, and 'I' < 'S').

Whereas ('UCI', 100) < ('UCI', 200) is True because at index 0 of these tuples, the string values are equal ('UCI' == 'UCI'), so we go to index 1, where we find the first different values, and 100 < 200.

Finally, ('UCI', 100) < ('UCI', 100, 'extra') is True because at index 0 these tuples the values are equal ('UCI' == 'UCI'); and at index 1 of these tuples the values are equal (100 == 100), so there are no different values; but the second tuple has a larger length, so it is greater than the first tuple.

Recall the sorting code from above.

```
votes = [('Charlie', 20), ('Able', 10), ('Baker' ,20), ('Dog', 15)]
for c,v in sorted(votes):
    print('Candidate', c, 'received', v, 'votes')
print(votes)
```

The reason that the values come out in the order they do (alphabetical order) in the code above is because the names that are in the first index in each 2-tuple are different and ensure the tuples are sorted alphabetically. Python never gets to looking at the second value in each tuple, because the first values (the candidate names) are always different.

Now, what if we don't want to sort by the natural tuple ordering. We can specify a "key" function that computes a key value for every value in what is being sorted, and the COMPUTED KEY VALUES ARE USED FOR COMPARISON, not the original values themselves. These are the "keys" for comparison.

See the by_vote_count function below; it takes a 2-tuple argument and returns only the second value in the tuple (recall indexes start at 0) for the key on which Python will compare. For the argument 2-tuple ('Baker' ,20) by_vote_count returns 20.

```
def by_vote_count(t):
    return t[1]            # remember t[0] is the first index, t[1] the second
```

So, when we sort with key=by_vote_count, we are telling the sorted function to determine the order of values by calling the by_vote_count function on each value: so in this call of sorted, we are comparing tuples based solely on their vote part, from index 1. So writing

```
votes = [('Charlie', 20), ('Able', 10), ('Baker' ,20), ('Dog', 15)]
for c,v in sorted(votes, key=by_vote_count):
    print('Candidate', c, 'received', v, 'votes')
```

produces

```
Candidate Able received 10 votes
Candidate Dog received 15 votes
Candidate Charlie received 20 votes
Candidate Baker received 20 votes
```

First, notice that by writing "key=by_vote_count" Python didn't CALL the by_vote_count function (there are no parentheses) it just passed its associated function object to the key parameter in the sorted function. Inside the sorted function, by_vote_count's function object automatically is called where needed, to compare two 2-tuples, to determine in which order these values will appear in the returned list.

Also, because Charlie and Baker both received the same number of votes, they both appear at the bottom, but they might appear in either order; the order is unspecified (equal values can appear in any order); here Charlie appears first.

Finally, notice that the tuples are printed in ascending order by votes; generally for elections we want the votes to be descending, so we can combine using key and reverse (with reverse=True) in the call to the sorted function.

```
votes = [('Charlie', 20), ('Able', 10), ('Baker' ,20), ('Dog', 15)]
for c,v in sorted(votes, key=by_vote_count, reverse=True):
    print('Candidate', c, 'received', v, 'votes')
```

which produces

```
Candidate Charlie received 20 votes
Candidate Baker received 20 votes
Candidate Dog received 15 votes
Candidate Able received 10 votes
```

Again, since the top two candidates both received the same number of votes, they could appear in any order: the order is unspecified.

Now, rather than define this simple by_vote_count function, we can use a lambda instead, and write the following.

```
...
for c,v in sorted(votes, key=(lambda t : t[1]), reverse=True):
    ...
```

So, now we don't have to write/name that extra by_vote_count function. Of course, if we did write it, we could reuse it wherever we wanted, instead of rewriting the lambda (but the lambdas are pretty small). By writing the lambda, the relevant code is all inside the call to sorted: we don't have to go look elsewhere for some function defintion. So, we now know how to use functions and lambdas in sorted.

Such a key function is used the same way when calling the sort method. If votes is the list shown at the beginning of this section (not the dictionary above), we can call votes.sort(key=(lambda t : t[1]), reverse=True) to sort this list, mutating it.

Another way to sort in reverse order (for integers) is to use the "negation" trick illustrated below (and omit reverse=True).

```
...
for c,v in sorted(votes, key=(lambda t : -t[1]) ):
    ...
```

Here we have negated the vote count part of the tuple, and removed reverse=True. So it is sorting from smallest to largest, but the key function returns the negative of the vote values (because that is what the lambda says to do). It is comparing -20, -10, -20, and -15 to sort. So here the biggest vote count corresponds to the smallest negative number (so the tuple with that number will appear first). The tuples appear in the order specified by the key functions: -20, -20, -15, -10 (smallest to largest). These negative values returned by the key function are used only to determine the order insides sorted; they do not appear in the results.

Finally, typically when multiple candidates are tied for votes, we want to print their names together (because they all have the same number of votes) but also in alphabetical order. We do this in Python by specifying a key function that returns a tuple in which the vote count is checked first and sorted in descending order; and only if the votes are equal will the names be checked and sorted in ascending order. Because we want the tuples in decreasing vote counts but increasing names, we cannot just use reverse=True: it would reverse both the vote AND name comparisons; we need to resort to the "negation trick" above and write

```
votes = [('Charlie', 20), ('Able', 10), ('Baker' ,20), ('Dog', 15)]
for c,v in sorted(votes, key=(lambda t : (-t[1],t[0])) ):
    print('Candidate',c,'received',v,'votes')
```

So it compares the 2-tuple keys (-20, 'Charlie'), (-10, 'Able'), (-20, 'Baker'), and (-15, 'Dog') when ordering the actual 2-tuples, and by what we have learned

(-20, 'Baker') < (-20, 'Charlie') < (-15, 'Dog') < (-10, 'Able')

which produces

```
Candidate Baker received 20 votes
Candidate Charlie received 20 votes
Candidate Dog received 15 votes
Candidate Able received 10 votes
```

So with this key function, the previous order is GUARANTEED, even though Baker and Charlie both received the same number (20) of votes, 'Baker' < 'Charlie' so that 2-tuple will appear first.

Finally, note that the lambda in the key ensures Python compares ('Charlie', 20) and ('Dog', 15) as if they were (-20, 'Charlie') and (-15, 'Dog'), so the first tuple will be less (and appear earlier) in the sorted list (the one with the highest votes has the lowest negative votes). And when Python compares ('Charlie', 20) and ('Baker' ,20) as if they were (-20, 'Charlie') and (-20, 'Baker') so the second tuple will be less and the tuple it was produced from will appear earlier (equal votes and then 'Baker' < 'Charlie').

So think of sorting

```
('Charlie', 20)   ('Able', 10)   ('Baker' ,20)   ('Dog', 15)
       |                |               |              |  using the key function
       |                |               |              |  it really compares
       V                V               V              V
(-20, 'Charlie')  (-10, 'Able')  (-20, 'Baker')  (-15, 'Dog')
```

which sorts to

```
(-20,'Baker')     (-20,'Charlie')   (-15,'Dog')    (-10,'Able')
       |                |               |              | actual values in list
       |                |               |              | (without key function)
       V                V               V              V
('Baker',20)      ('Charlie', 20)   ('Dog',15)     ('Able', 10)
```

so the order of the actual list returned by sorted is

[('Baker', 20), ('Charlie', 20), ('Dog', 15), ('Able', 10)]

which is sorted by decreasing vote, with equal votes sorted alphabetically increasing by name.

Our ability to use this "negation trick" works in SOME cases, but unfortunately NOT IN ALL cases: we can sort arbitarily using this trick exactly when "all non-numerical data is sorted in the same way" (all increasing or all decreasing). In such cases, we can negate any numerical data, if we need to sort it decreasing. So, we can sort the above example because we are sorting only one non-numerical datum. By the requirement, all non-numerical data (there is only one on this example, the name) is sorted the same way (increasing).

Using the "negation trick" is therefore not generalizable to all sorting tasks: for example, if we had a list of 2-tuples containing strings of candidates and strings of the states they are running in

```
  [('Charlie', 'CA'), ('Able', 'NY'), ('Baker' ,'CA'), ('Dog', 'IL')]
```

there is no way to use the "negation trick" to sort them primarily by decreasing state, and secondarily by increasing name (when two candidates come from the same state). Here it is not the case that "all non-numerical data is sorted the same way": each string data is sorted differently. In this case the

sorted list would be

```
  [('Able', 'NY'), ('Dog', 'IL'), ('Charlie', 'CA'), ('Baker' ,'CA')]
```

even though we wanted it to be (note the last two names, from the same state)

```
  [('Able', 'NY'), ('Dog', 'IL'), ('Baker', 'CA'), ('Charlie' ,'CA')]
```

We cannot apply the "negation trick" to either of the strings: we cannot negate strings at all. But we can produce a list in this specified order by calling sorted multiple times, as is illustrated below.

Arbitrary sorting: multiple calls to sorted with "stable" sorting

Python's sorted function (and sort method) are "stable". This property means that "equal" values (decided naturally, or with the key function supplied) keep their same "relative" order (left-to-right positions) in the data being sorted.

For example, assume that db is a list of 2-tuples, each specifying a student: index 0 is a student's name; index 1 is that student's grade.

```
  db = [('Bob','A'),    ('Mary','C'), ('Pat','B'), ('Fred','B'), ('Gail','A'),
        ('Irving','C'), ('Betty','B'), ('Rich','F')]
```

If we call

```
  sorted(db, key = lambda x : x[1])   # sorted by index 1 only: their grade
```

Python returns the following list, whose 2-tuples are sorted by grade. Because sorting is "stable", all 2-tuples with the same grade (equal values by the key function) keep their same relative order (left-to-right positions) in the list.

```
  [('Bob','A'),  ('Gail','A'),   ('Pat','B'), ('Fred','B'), ('Betty','B'),
   ('Mary','C'), ('Irving','C'), ('Rich','F')]
```

Notice that...

(1) in the original list, the students with 'A' grades are 'Bob' and 'Gail', with 'Bob' to the left of 'Gail'. In the returned list, 'Bob' is also to the left of 'Gail'.

(2) in the original list, the students with 'B' grades are 'Pat', 'Fred', and 'Betty', with 'Pat' to the left of 'Fred' and 'Fred' to the left of 'Betty'. In the returned list, 'Pat' is to the left of 'Fred' and 'Fred' is to the left of 'Betty'.

(3) in the original list, the students with 'C' grades are 'Mary' and 'Irving', with 'Mary' to the left of 'Irving'. In the returned list, 'Mary' is also to the left of 'Irving'.

(4) there is only one student with an 'F' grade.

Now, suppose that we wanted to sort this list so that primarily the 2-tuples are sorted DECREASING by grade ('F's, then 'D's, then 'C's, then 'B's, then 'A's); and for students who have equal grades, the 2-tuples are sorted INCREASING alphabetically by student name. We can accomplish this task by calling sorted twice, by writing

```
  sorted( sorted(db, key=(lambda x : x[0])), key=(lambda x : x[1]), reverse=True )
```

The inner call to sorted produces the list

```
  [('Betty','B'), ('Bob','A'), ('Fred','B'), ('Gail','A'), ('Irving','C'),
   ('Mary','C'),  ('Pat','B'), ('Rich','F')]
```

which is sorted INCREASING by the student's name: all names are distinct (different), so stability is irrelevant here.

The outer call to sorted, using the "sorted by name" db list as an argument, produces the list

```
[('Rich','F'), ('Irving','C'), ('Mary','C'), ('Betty','B'), ('Fred','B'),
 ('Pat','B'),  ('Bob','A'),    ('Gail','A')]
```

which is sorted DECREASING by the student's grade: for equal grades, the stability property ensures that all names are still sorted INCREASING by the student's name (because the argument is sorted that way). Here, the 'B' students are in the alphabetical order 'Betty', 'Fred', and 'Pat'. Note that we could simplify this code to just

```
sorted( sorted(db), key=(lambda x : x[1]), reverse = True )
```

because for the inner call to sorted, no key function is the same as the key function that specifies sorting the 2-tuples in the natural way (alphabetically by name). We could use a temporary variable and write this code as

```
temp = sorted(db)
sorted( temp, key = lambda x : x[1], reverse = True )
```

Thus, we can sort complex structures in any arbitrary way (some data increasing, some data decreasing) by calling sorted multiple times on it. The LAST call will dictate the primary order in which it is sorted; each preceding/inner call dictates how the data is sorted if values specified by the outer orders are equal.

Experiment with these various forms of sorting, using reverse, keys, the negation trick, and multiple calls to sort.

Finally, to solve the original problem above (with candidates and their states: also involving two strings sorted primarily DECREASING on state and INCREASING on name)

```
db  = [('Charlie', 'CA'), ('Able', 'NY'), ('Baker' ,'CA'), ('Dog', 'IL')]
```

we would call

```
sorted( sorted(db), key=(lambda x : x[1]), reverse = True )
```

Bottom line on sorting:

To achieve our sorting goal, if we can call sorted once, using a complicated key (possibly involving the "negation trick") and reverse, that is the preferred approach: it is shortest/clearest in code and fastest in computer time. But if we are unable to specify a single key function and reverse that do the job, we can always call sorted multiple times, using multiple key functions and multiple reverses, sometimes relying on the "stability" property to achieve our sorting goal.

We now have a general tool bag for sorting all types of information. Sorting will be heavily tested in Quiz #1, Programming Assignment #1, and In-Lab #1.

The print function

Notice how the sep and end parameters in print help control how the printed values are separated and what is printed after the last one. Recall that print can have any number of arguments (we will see how this is done in Python soon), and it prints the str(...) of each parameter. By default, sep=' ' (space) and end='\n' (newline). So the following

```
print(1,2,3,4,sep='--',end='/')
print(5,6,7,8,sep='x',end='**')
```

prints

```
1--2--3--4/5x6x7x8**
```

Also recall that all functions must return a value. The print function returns the value None: this function serves as a statement: it has an "effect" (of displaying information in the console window; we might say changing the state of the console) but returns no useful "value"; but all functions must return a value, so this one returns None. Recall that the sort method on lists mutated a list (its primarly action) but also had to return a value, so returned None.

Sometimes we use sep='' to control spaces more finely. In this case we must put in all the spaces ourselves. If we want to separate only 2 and 3 by a space, we write

```
print(1,2,' ',3,sep='')
```

which prints

```
12 3
```

Still, other times we can concatenate values together into one string (which requires us to explicitly use the str function on the things we want to print).

```
x = 10
print('Your answer of '+str(x)+' is too high.'+'\nThis is on the next line')
```

Note the use of the "escape" sequence \n to generate a new line. It doesn't print a '\' followed by an 'n'; Python interprets '\n' to be the newline character: the remaining output starts at the beginning of the next line.

Finally, we can also use the very general-purpose .format function. This function is illustrated in Section 6.1.3 in the documentation of The Python Standard Library. The following two statements print equivalently: for a string with many format substitutions, I prefer the form with a name (here x) in the braces of the string and as a named parameter in the arguments to format.

```
print('Your answer of {} is too high\nThis is on the next line'.format(10))

print('Your answer of {x} is too high\nThis is on the next line'.format(x=10))
```

In Python 3.6 and beyond we can use f-strings (formatted strings; google PEP 498) to solve this problem even more simply.

```
print(f'Your answer of {x} is too high\nThis is on the next line')
```

Here, any expression in {} is evaluated and turned into a string embedded in the f-string. We could write {x+1} or any other expression instead of just the x in the {}.

String/List/Tuple (SLT) slicing:

SLTs represent sequences of indexable values, whose indexes start at index 0, and go up to -but do not include- the length of the SLT (which we can compute using the len function).

1) Indexing: We can index a SLT by writing SLT[i], where i is in the range 0 to len(SLT)-1 inclusive, or i is negative and i is in the range -len(SLT) to -1 inclusive: if an index is not in these ranges, Python raises an exception. Note SLT[0] is the value store in the first index, SLT[-1] is the value stored in the last index, SLT[-2] is the value stored in the 2nd to last index.

2) Slicing: We can specify a slice by SLT[i:j] which includes SLT[i] followed by SLT[i+1], ... SLT[j-1]. Slicing a string produces another string, slicing a list produces another list, and slicing a tuple produces another tuple). The resulting structures has j-i elements if indexes i through j are in the SLT (or 0 if j-i is <= 0); for this formula, both indexes must be non-negative or both positive; if they are different, convert the negative

```
    to it non-negative (or the non-negative to its negative) equivalent.

    If the slice contain no values it is empty (an  empty string, an empty list,
     and empty tuple).

    If the first index come before index 0, then index 0 is used; if the second
    index comes after the biggest index, then index len(SLT) is used. Finally,
    if the first index is omitted it is 0; if the second index is omitted it is
    len(SLT).

    Here are some examples

s = 'abcde'
x = s[1:3]
print (x)     # prints 'bc' which is 3-1 = 2 values
x = s[-4:-1]
print (x)     # prints 'bcd' same as s[1:4] which is 4-1 = 3 values
x = s[1:-2]
print (x)     # prints 'bc' same as s[1:3] which is 3-1 = 2
x = s[-4:-2]
print (x)     # prints 'bc' same as s[1:3] which is 3-1 = 2 values
x = s[1:10]
print (x)     # prints 'bcde' which is len(x)-1 = 4 values

    likewise

s = ('a','b','c','d','e')
x = s[1:3]
print (x)     # prints ('b','c') which is 3-1 = 2 values
x = s[-4:-1]
print (x)     # prints ('b','c','d') which is -1-(-4) = 3 values

    s[:i] is index 0 up to but not including i (can be positive or negative)
    s[i:] is index i up to and including the last index; so s[-2:] is the last
          two values.

3) Slicing with a stride: We can specify a slice by SLT[i:j:k] which includes
   SLT[i] followed by SLT[i+k], SLT[i+2k] ... SLT[j-1]. This follows the rules
   for slicing too, and allows negative numbers for indexing.

s = ('a','b','c','d','e')
x = s[::2]
print (x)     # prints ('a','c','e')
x = s[1::2]
print (x)     # prints ('b','d')

x = s[3:1:-1]
print (x)     # prints ('d','c')
x = s[-1:1:-1]
print (x)     # prints ('e','d','c')

  When the stride is omitted, it is +1. If the stride is negative, if the first
  index is omitted it is len(SLT); if the last index is omitted it is -1.
  Compare these to the values if omitted above in part 2, which assumed a
  positive stride.

Experiment with various slices of strings (the easiest to write) to verify you
understand what results they produce.

--------------------------------------------------------------------------------

Conditional statement vs. Conditional expression

Python has an if/else STATEMENT, which is a conditional statement. It also has a
conditional EXPRESSION that uses the same keywords (if and else), which while
not as generally useful, sometimes is exactly the right tool to simplify a
programming task.

A conditional statement uses a boolean expression to decide which indented
```

block of statements to execute; a conditional expression uses a boolean expression to decide which one of exactly two other expressions to evaluate: the value of the evaluated expression is the value of the conditional expression.

The form of a conditional expression is

```
resultT if test else resultF
```

This says, the expression evaluates to the value of resultT if test is True and the value of resultF if test is False; first it evaluates test, and then evaluates either resultT or resultF (but only one, not the other) as necessary. Like other short-circuit operators in Python (do you know which?) it evaluates only the subexpressions it needs to determine the result.

I often write conditional expresions inside parentheses for clarity (as I did for lambdas; sometimes the parentheses are required, as with lambdas). See the examples below.

Here is a simple example. We start with a conditional statement, which always stores a value into min: either x or y depending on whether x <= y. Note that regardless of the test, min is bound to some value.

```
if x <= y:
    min = x
else:
    min = y
```

We can write this using a simpler conditional expression, capturing the fact that we are always storing into min, and just deciding which value to store in it.

```
min = (x if x <= y else y)
```

Not all conditional statements can be converted into conditional expressions; typically only simple ones can: ones with a single statement in their indented blocks. But using conditional expressions in these cases simplifies the code even more. So attempt to use conditional expression, but use good judgement after you see what the code looks like. Write the simplest looking code.

Here is another example; it always prints x followed by some message

```
if x % 2 == 0:
    print(x,'is even')
else:
    print(x,'is odd')
```

We can rewrite it as as calling print with a conditional expression inside deciding what string to print at the end.

```
    print(x, ('is even' if x%2 == 0 else 'is odd'))
```

We can also write it as a one argument print using concatenation:

```
    print(str(x) + ' is ' + ('even' if x%2 == 0 else 'odd'))
```

Note that conditional expressions REQUIRE using BOTH THE if AND else KEYWORDS, along with the test boolean expression and the two expressions needed in the cases where the test is True or the test is False. Some conditional statements use just if (and no else nor elif).

As an example of f-strings, we can write this as

```
    print(f"{x} is {'even' if x%2 == 0 else 'odd'}")
```

because we can write a conditional EXPRESSION inside {}, because we can put any expression there. Note I had to write the outside string using " to specify 'even' and 'odd' as strings in the f-string.

The else: block-else option in for/while loops

For and while looping statements are described as follows. The else: block-else is optional, and not often used. But we will explore its meaning here.

```
for_statement   <= for index(es) in iterable:
                       block-body
                   [else:
                       block-else]

while_statement <= while <bool-expression>:
                       block-body
                   [else:
                       block-else]
```

Here are the semantics of else: block-else.

> If the else: block-else option appears, and the loop terminated normally, (not with a break statement) then execute block-else.

Here is an example that makes good use of the else: block-else option. This code prints the first/lowest value (looking at the values 0 to 100 inclusive) for which the function special_property returns True (and then breaks out of the loop); otherwise it prints that no value in this range had this property: so it prints exactly one of these two messages. Note you cannot run this code, because there is no special_property function: I'm using it for illustration only.

```
for i in irange(100):
    if special_property(i):
        print(i,'is the first value with the special property')
        break
else:
    print('No value in the range had the special property')
```

Without the else: block-else option, the simplest code that I can write that has equivalent meaning is as follows.

```
found_one = False
for i in irange(100):
    if special_property(i):
        print(i,'is the first with the special property')
        found_one = True
        break
if not found_one:
    print('No value in the range had the special property')
```

This solution requires an extra name (found_one), an assignment to set and reset the name, and an if statement. Although I came up with the example above, I have not used the else: block-else option much in Python. Most programming languages that I have used don't have this special feature, so I'm still exploring its usefulness. Every so often I have found an elegant use of this construct.

Can you predict what would happen if I removed the break statement in the bigger code above? Run the code to check your answer.

Argument/Parameter Matching (leaves out **kargs, discussed later)

Recall that when functions are called, Python first evaluates the arguments, then matches/binds them with parameters, and finally executes the body of the function using the names of parameters (bound to the values of arguments).

Let's explore the argument/parameter matching rules. First we classify arguments and parameters, according to the options that they include. Remember

that arguments appear in function CALLS and parameters appear in function HEADERS (the first line in a function defintion).

Arguments
  positional argument: an argument NOT preceded by the name= option
  named   argument: an argument preceded by the name= option

Parameters
  name-only parameter   :a parameter not followed by =default argument value
  default-argument parameter:a parameter followed by =default argument value

When Python calls a function, it must define every parameter name in the function's header (binding each to the argument value object matching that parameter's name, just like an assignment statement). In the rules below, we will learn exactly how Python matches arguments to parameters according to three criteria: positions, parameter names, and default arguments for parameter names. We will also learn how to write functions that can receive an arbitrary number of arguments (so, for example, we can write our own print function).

Here is a concise statement of Python's rules for matching arguments to parameters. The rules are applied in this order (e.g., once you reach M3 we cannot go back to M1).

M1. Match positional argument values in the call sequentially to the parameters named in the header's corresponding positions (both name-only and default-argument parameters are OK to match). Stop when reaching any named argument in the call, or the * parameter (if any) in the header.

M2. If matching a * parameter in the header, match all remaining positional argument values to it. Python creates a tuple that stores all these arguments. The parameter name (typically args) is bound to this tuple.

M3. Match named-argument values in the call to their like-named parameters in the header (both name-only and default-argument parameters are OK).

M4. Match any remaining default-argument parameters in the header (unmatched by rules M1 and M3) with their specified default argument values.

M5. Exceptions: If at any time (a) an argument cannot match a parameter (e.g., a positional-argument follows a named-argument) or (b) a parameter is matched multiple times by arguments; or if at the end of the process (c) any parameter has not been matched or (d) if a named-argument does not match the name of a parameter, raise an exception: SyntaxError for (a) and TypeError for (b), (c), and (d). These exceptions report that the function call does not correctly match its header by these rules.

[When we examine a **kargs as a parameter, we will learn what Python does when there are extra named arguments in a function call: names besides those of parameters: preview: it puts all remaing named arguments in a dictionary, with their name as the key and their value associated with that key). The parameter name (typically kargs or kwargs) is bound to this dictionary.]

When this argument-parameter matching process if finished, Python defines, (in the function's namespace), a name for every parameter and binds each to the unique argument it matched using the above rules. Passing parameters is similar to performing a series of assignment statements between parameter names and their matching argument values.

If a function call raises no exception, these rules ensure that each parameter in the function header matches the value of exactly one argument in the function call. After Python binds each parameter name to its argument, it executes the body of the function, which computes and returns the result of calling the function.

Here are some examples of functions that we can call to explore the rules for argument/parameter matching specified above. These functions just print their parameters, so we can see the arguments bound to them (or see which exception they raise). Certainly you should try others to increase your understanding.

```
def f(a,b,c=10,d=None): print(a,b,c,d)
def g(a=10,b=20,c=30) : print(a,b,c)
def h(a,*b,c=10)      : print(a,b,c)

Call               | Parameter/Argument Binding (matching rule)
-------------------+-------------------------------------------
f(1,2,3,4)         | a=1, b=2, c=3, d=4(M1)
f(1,2,3)           | a=1, b=2, c=3(M1); d=None(M4)
f(1,2)             | a=1, b=2(M1); c=10, d=None(M4)
f(1)               | a=1(M1); c=10, d=None(M4);
                       TypeError(M5c: parameter b not matched)
f(1,2,b=3)         | a=1, b=2(M1); b=3(M3); c=10, d=None(M4)
                     TypeError(M5b: parameter b matched twice)
f(d=1,b=2)         | d=1, b=2(M3); c=10(M4);
                       TypeError(M5c: parameter a not matched)
f(b=1,a=2)         | b=1, a=2(M3); c=10, d=None(M4)
f(a=1,d=2,b=3)     | a=1, d=2, b=3(M3); c=10(M4)
f(c=1,2,3)         | c=1(M3);
                       SyntaxError(M5a:2 is positional argument)

g()                | a=10, b=20, c=30(M4)
g(b=1)             | b=1(M3); a=10, c=30(M4)
g(a=1,2,c=3)       | a=1(M3);
                       SyntaxError(M5a:2 is positional argument)

h(1,2,3,4,5)       | a=1(M1); b=(2,3,4,5)(M2), c=10(M4)
h(1,2,3,4,c=5)     | a=1(M1); b=(2,3,4)(M2), c=5(M3)
h(a=1,2,3,4,c=5)   | a=1(M3);
                       SyntaxError(M5a:2 is positional argument))
h(1,2,3,4,c=5,a=1)| a=1(M1); b=(2,3,4)(M2); c=5(M3);
                        TypeError(M5b:a matched twice)
```

Here is a real but simple example of using *args, showing how the max function is implememented in Python; we dont' really need to write this function because it is in Python already, but here is how it is written in Python. We will cover raising exceptions later in this lecture note, so don't worry about that code.

```
def max(*args) :           # Can refer to args inside; it is a tuple of values
    if len(args) == 0:
        raise TypeError('max: expected >=1 arguments, got 0')

    answer = None
    for i in args:
        if answer == None or i > answer:
            answer = i
    return answer

print(max(3,-4, 2, 8)) # max with many arguments; prints 8
```

In fact, the real max function in Python can take either (a) any number of arguments or (b) one iterable argument. It is a bit more subtle to write correctly to handle both parameter structures, but here is the code.

```
def max(*args) :           # Can refer to args inside; it is a tuple of values
    if len(args) == 0:
        raise TypeError('max: expected >=1 arguments, got 0')

    if len(args) == 1:     # Assume that if max has just one argument then
        args = args[0]     #   it's iterable, so take the max over its values

    answer = None
    for i in args:
        if answer == None or i > answer:
            answer = i
    return answer

l = (3,-4, 2, 8)
print(max(l))          # max with one iterable argument; prints 8
```

Finally, because of this approach computing max(3) raises an exception, because Python expects a single argument to be iterable. It might be reasonable in this case to return 3, but that is not the semantics (meaning) of the max function built into Python.

Here is another real example of using *args, where I show how the print function is written in Python. The myprint calls a very simple version of print, just once at the end, to print only one string that it builds from args along with sep and end; it prints the same thing the normal print would print with the same arguments. Notice the use of the conditional if in the first line, to initialize s to either '' or the string value of the first argument.

```
def myprint(*args, sep=' ', end='\n'):
    s = (str(args[0]) if len(args) >= 1 else '') # handle 1st (if there) special
    for a in args[1:]:                           # all others come after sep
        s += sep + str(a)
    s += end                                     # end at the end
    print(s,end='')                              # print the entire string s

myprint('a',1,'x')                  # prints a line
myprint('a',1,'x',sep='*',end='E') # prints a line but stays at end
myprint('a',1,'x')                  # continues at end of previous line
```

Together when executed, these print

```
a 1 x
a*1*xEa 1 x
```

This is what would be printed if the print (not myprint) function was called.

Constructors operating on Iterable values to Construct Data

List, Tuples, Sets:

Python's "for" loops allow us to iterate through all the components of any iterable data. We can even iterate through strings: iterating over their individual characters. Later in the quarter, we will study iterator protocols in detail, both the special iter/__iter__ and next/__next__ methods in classes, and generators (which are very very similar to functions, with a small but powerful twist). Both will improve our understanding of iterators and also allow us to write our own iterable data types (classes) easily.

Certainly we know about using "for" loops and iterable data (as illustrated by lots of code above). What I want to illustrate here is how easy it is to create lists, tuples, and sets from anything that is iterable by using the list, tuple, and set constructors (we'll deal with dict constructors later in this section). For example, in each of the following the constructor for the list/tuple/set objects iterates over the string argument to get the 1-char strings that become the values in the list/tuple/set object.

```
l = list ('radar') then l is ['r', 'a', 'd', 'a', 'r']
t = tuple('radar') then t is ('r', 'a', 'd', 'a', 'r')
s = set  ('radar') then s is {'a', 'r', 'd'} or {'d', 'r', 'a'} or ...
```

Note that lists/tuples are ORDERED, so whatever the iteration order of their iterator argument is, the values in the list/tuple will be the same order. Contrast this with sets, which have (no duplicates and) no special order. So, set('radar') can print its three different values in any order.

Likewise, since tuples/sets are iterable, we can also compute a list from a list, a list from a a tuple, or a list from a set. Using l, t, and s from above.

```
list(t) which is ['r', 'a', 'd', 'a', 'r']
list(s) which is ['r', 'd', 'a']  assuming s iterates in the order 'r', 'd', 'a'
list(l) which is ['r', 'a', 'd', 'a', 'r']
```

The last of these iterates over the list to create a new list with the same

values: note that 1 is list(1) is False, but 1 == list(1) is True: there are two different lists, but they store the same contents (see the next section on "is" vs "==" for more details).

Likewise we could create a tuple from a list/set, or a set from a list/tuple. All the constructors handle iterable data, producing a result of the specified type by iterating over their argument.

Note that students sometimes try to use a list constructor to create a list with one value. They write list(1), by Python responds with "TypeError: 'int' object is not iterable" because the list constructor is expecting an iterable argument. The correct way to specify this list with one value is [1]. Likewise for tuples, sets, and dictionaries.

We have seen that opened file objects are also iterable, where each line in the file is iterated over. So the statement

```
contents =  list(open('test.txt'))
```

will resut in contents being a list of all the lines in the file (with the special newline ('\n') character at the end of each line).

Program #1 will give you lots of experience with these data types and when and how to use them. The take-away now is it is trivial to convert from one of these data types to another, because the constructors for their classes all allow iterable values as their arguments, and all these data types (and strings as well) are iterable (can be iterated over).

Dictionary Constructors:

Before leaving this topic, we need to look at how dictionaries fit into the notion of iterable. There is not just ONE way to iterate through dictionaries, but there are actually THREE ways to iterate through dictionaries: by keys, by values, and by items (each item a 2-tuple with a key folowed by its associated value). Each of these is summoned by a method name for dict, and the methods are named the same: keys, values, and items.

So if we write the following to bind d to a dict (we will discuss this "magic" constructor soon)

```
d = dict(a=1,b=2,c=3,d=4,e=5) # the same as d = {'a':1,'b':2,'c':3,'d':4,'e':5}
```

Then we can create lists of three aspects of the dict:

```
list(d.keys  ()) is like ['c', 'b', 'a', 'e', 'd']
list(d.values()) is like [3, 2, 1, 5, 4]
list(d.items ()) is like [('c', 3), ('b', 2), ('a', 1), ('e', 5), ('d', 4)]
```

I said "is like" because sets and dicts are NOT ORDERED: in the first case we get a list of the keys; in the second a list of the values; in the third a list of item tuples, where each tuple contains one key and its associated value. But in all three cases, the list's values can appear in ANY ORDER.

Note that the keys in a dict are always unique, but there might be duplicates among the values: try the code above with d = dict(a=1,b=2,c=1). Items are unique because they contain keys (which are unique).

Also note that if we iterate over a dict without specifying how, it is equivalent to specifyihg d.keys(). That is

list(d) is the same as list(d.keys()) which is like ['a', 'c', 'b', 'e', 'd']

One way to construct a dict is to give it an iterable, where each value is either a 2-tuple or 2-list: a key followed by its associated value. So, we could have written any of the following to initialize d:

```
d = dict( [['a', 1], ['b', 2], ['c', 3], ['d', 4], ['e', 5]] ) #list of  2-list
d = dict( [('a', 1), ('b', 2), ('c', 3), ('d', 4), ('e', 5)] ) #list of  2-tuple
```

```
d = dict( (['a', 1], ['b', 2], ['c', 3], ['d', 4], ['e', 5]) ) #tuple of 2-list
d = dict( (('a', 1), ('b', 2), ('c', 3), ('d', 4), ('e', 5)) ) #tuple of 2-tuple

or, even (a tuple that has a mixture of 2-tuples and 2-lists in it)

d = dict( (('a', 1), ['b', 2], ('c', 3), ['d', 4], ('e', 5)) )

or even (a set of 2-tuples; we cannot have a set of 2-list (see hashable below)

d = dict( {('a', 1), ('b', 2), ('c', 3), ('d', 4), ('e', 5)} )

The bottom line is that a positional dict argument must be iterable, and each
value in the iterable must have 2 values (e.g., a 2-list or 2-tuple) that
represent a key followed by its associated value.

We can also combine the two forms writing
d = dict( {('a', 1), ('b', 2), ('c', 3), ('d', 4), ('e', 5)}, f=6, g=7)

Finally, if we wanted to construct a dict using the keys/values in another
dict, here are two easy ways to do it

d_copy = dict(d)

or

d_copy = dict(d.items())

In both cases the dict constructor creates a new dictionary by iterating through
the (key,value) 2-tuples.

--------------------------------------------------------------------------------

Sharing/Copying:  is vs. ==
  (one more time: see Binding (and Drawing Names and their associated Objects)

It is important to understand the fundamental difference between two names
sharing an object (bound to the same object) and two names referring/bound to
"copies of the same object". Note that if we mutate a shared object, both names
"see" the change: both are bound to the same object which has mutated. But if
they refer to different copies of an object, only one name "sees" the change.

Note the difference between the Python operators is and ==. Both return boolean
values. The first asks whether two references/binding are to the same object
(the is operator is called the (object)identity operator); the second asks only
whether the two objects store the same values. See the different results
produced for the example below. Also note that if x is y is True, then x == y
must be True too: an object ALWAY stores the same values as itself. But if
x == y is True, x is y may or may not be True.

For example, compare execution of the following scripts: the only difference
is the second statement in each: y = x vs. y = list(x)

x = ['a']
y = x           # Critical: y and x share the same reference
print('x:',x,'y:',y,'x is y:',x is y,'x == y:',x==y)
x [0] = 'z'     # Mutate x (could also append something to it)
print('x:',x,'y:',y,'x is y:',x is y,'x == y:',x==y)

This prints
x: ['a'] y: ['a'] x is y: True x == y: True
x: ['z'] y: ['z'] x is y: True x == y: True

x = ['a']
y = list(x)     # Critical: y refers to a new list with the same contents as x
print('x:',x,'y:',y,'x is y:',x is y,'x == y:',x==y)
x [0] = 'z'     # Mutate x (could also append something to it: x+)
print('x:',x,'y:',y,'x is y:',x is y,'x == y:',x==y)

This prints
```

```
x: ['a'] y: ['a'] x is y: False x == y: True
x: ['z'] y: ['a'] x is y: False x == y: False
```

You might have learned about the id function: it returns a unique integer for any object. It is often implemented to return the first address in memory at which an object is stored, but there is no requirement that it returns this int. Checking "a is b" is equivalent to checking "id(a) == id(b)" but the first way to do this check (with the is operator) is preferred (and is faster).

Finally there is a copy module in Python that defines a copy function: it copies some iterable without us having to specify the specific constructor (like list, set, tuple, or dict).

So we can import it as: from copy import copy

Assuming x is a list, we can replace y = list(x) by y = copy(x).
Likewise, if x is a dict we can replace y = dict(x) by y = copy(x).

We could also just: import copy (the module) and then write y = copy.copy(x) but it is clearer in this case to write "from copy import copy". Here is a simple implementation of copy:

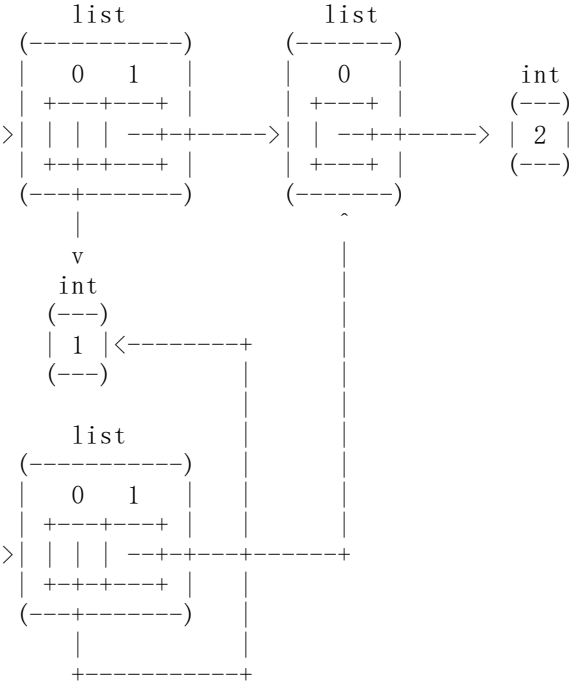
```
def copy(x):
    return type(x)(x)
```

which finds the type of x, then calls it as a constructor to construct a new object of that type, intialized by x: iterating over all the values in x, as we say in examples list list('abc') producing ['a', 'b', 'c'].

Note that copying in all the ways that we have discussed is SHALLOW. That means the "copy" is a new object, but that object stores all the references from the "object being copied". For example, if we write

```
x = [1, [2]]
y = list(x) # or y = copy(x) or y = x[:]
```

then we would draw the following picture for these assignments.



Here, x and y refer to DIFFERENT lists: but all the references in x's list are identical to the references in y's list. Using the 'is" operator we can state that (1) x[0] is y[0] == True, and (2) x[1] is y[1] == True.

This means if we now write x[1][0] = 'a', then print(y[1][0]) prints 'a' too,

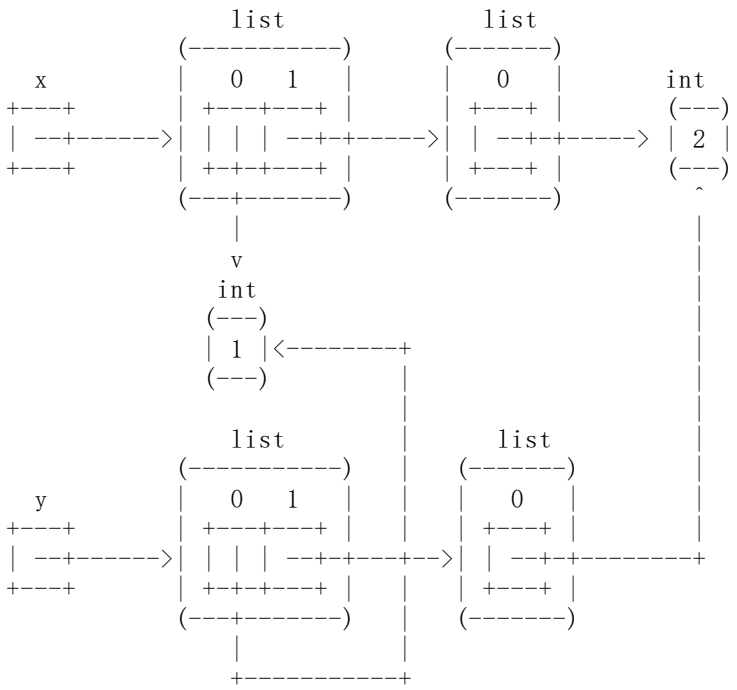
because x[1] and y[1] refer to the same list.

We will discuss how to do DEEP copying when we discuss recursion later in the course; it is implemented in the copy module by the function named deepcopy, which does a deep copy of an object, by constucting an object of that type and populating it with deep copies of all the objects in the original object.

But, if we wrote

```
from copy import deepcopy
x = [1, [2]]
y = deepcopy(x)
```

then we would draw the following picture for these assignments.



In a deep copy, all the mutable objects referred to are copied, but the immutable ones referred to are not copied.

For more information on mutable, see the next section.

Hashable vs. Mutable and how to Change Things:

Python uses the term Hashable, which has the same meaning as Immutable. So hashable and mutable are OPPOSITES: You might see this message relating to errors when using sets with UNHASHABLE values or dicts with UNHASHABLE keys: since hashable means immutable, then un-hashable means un-immutable which simplifies (the two negatives cancel) to mutable. So unhashable means the same as mutable. So

```
hashable    means the same as immutable
unhashalble means the same as mutable
```

Here is a quick breakdown of standard Python types

```
Hashable/immutable: numeric values, strings, tuples containing
                      hashable/immutable data, frozenset
mutable/unhashable: list, sets, dict
```

The major difference between tuples and lists in Python is the former is (mostly) hashable/immutable and the later is not. I say mostly because if a tuple contains mutable data, you can mutate the mutable data part. So

```
x = (1, 2, [3, 4])
x[2][0] = 'a'
```

```
  print(x)
```

prints

```
  (1, 2, ['a', 4])
```

So you have mutated the tuple because you mutated the list stored in index 2 of the tuple. On the other hand, you cannot append to a tuple, nor store a new value in one of its indexes (x[0] = 10 is not allowed).

So technically, a tuple storing hashable/immutable values is hashable/immutable, but a tuple storing unhashable/mutable values is unhashable/mutable. So if some other datatype (e.g., values in a set, or keys in a dictionary) needs to be hashable/immutable, use a tuple (storing hashable/immutable values) to represent its value, not a list. Thus we cannot say, "Any value whose type is a tuple is hashable/immutable." Instead we must say, "A value whose type is a tuple is hashable/immutabile when the tuple stores only hashable/immutable values."

A frozenset can do everything that a set can do, but doesn't allow any mutator methods to be called (so we cannot add a value to or delete a value from a frozenset). Thus, we can use a frozen set as a value in a set or a key in a dictionary.

The constructor for a frozenset is frozenset(...) not {}. Note that once you've constructed a frozen set you cannot change it (because it is immutable). If you have a set s and need an equivalent frozenset, just write frozenset(s).

The function hash takes an argument that is hashable (otherwise it raises TypeError, with a message about a value from an unhashable type) and returns an int.

We will study hashing towards the end of the quarter: it is a technique for allowing very efficient operations on sets and dicts. ICS-46 (Data Structures) studies hash tables in much more depth, in which you will implement the equivalent of Python sets and dicts by using hash tables.

Interlude: unique objects

Small integer objects are unique in Python. If we assign x = 1 and y = 1, then x is y evaluates to True (but you should use == and != for comparing integers). Likewise, if we write x += 1 and y *= 2 then x is y evaluates to True. But if we write x = 10**100 and y = 10**100. then x is y is False (but x == y is True, as expected).

To save space, Python allocates only one object for each small int. When a small int is computed, Python first looks to see if an object with that value already exists, and if it does, returns a reference to it; if it doesn't, it makes a new object storing that values and returns a reference to it. This is a tradeoff that minimizes the space (occupied by objects) but increases the time to do computations (by having to first look to see if an object for a value already exists).

Generally, because ints are immutable, there are no bad consequences in sharing objects in this way: once a name is bound to an int object, that int object will always represent the same value, because ints are immutable.

Again, constrast this with writing lists, which always allocate new objects. If we write

```
x = ['a']
y = ['a']
```

Then x is y evaluates to False. Because lists are mutable, this is the only correct way to implement lists.

The situation with strings (also immutable) is similar but more complicated. Whether "is" operating on two strings with the same characters is True depends

on things like the length of the strings and whether the string was computed or entered by the user in the console. If we write

```
x = 'ab'
y = 'a'+'b'
z = input('Enter string') # and enter the string ab when prompted
```

x is y evaluates to True, but x is z evaluates to False. As with integers, it is best to compare strings with == and != and not use the "is" or "is not" operators on them.

End of 2nd Lecture on this material

Comprehensions: list, tuple, set, dict

List, Tuple, Set Comprehensions:

Comprehensions are compact ways to create complicated (but not too complicated) lists, tuples, sets, and dicts. That is, they compactly solve some problems but cannot solve all problems (for example, we cannot use them to mutate values in an existing data structure, just to create values in a new data structure). The general form of a list comprehension is as follows, where f means any function using var (or expression using var: we can also write just var there because a name by itself is a very simple expression) and p means any predicate (or bool expresssion) using var.

```
[f(var,...) for var in iterable if p(var,...)]
```

Meaning: collect together into a list (list because of the outer []) all of f(var,...) values, for var taking on every value in iterable, but only collect an f(var,...) value if its corresponding p(var,...) is True.

For tuple or set comprehensions, we would use () and {} as the outermost grouping symbol instead of []. We'll talk about dicts later in this section: they use also use {} but also include a : inside (separating keys from values) to be distinguised from sets, which use {} without any such : inside.

Note that the "if p(var,...)" part is optional, so we can also write the simplest comprehensions as follows (in which case it has the same meaning as p(var,...) always being True).

```
[f(var,...) for var in iterable]
```

   which has the same meaning as

```
[f(var,...) for var in iterable if True]
```

for example

```
x = [i**2 for i in irange(1,10) if i%2==0]      # note: irange not range
print(x)
```

prints the squares of all the integers from 1 to 10 inclusive, but only if the integer is even (computed as leaving a remainder of 0 when divided by 2). Run it. Change it a bit to get is to do something else. Here is another example

```
x = [2*c for c in 'some text' if c in 'bcdfghjklmnpqrstvwxz']
print(x)
```

which prints a list with strings with doubled characters for all the consonants (no aeiouy -or spaces for that matter) in the string 'some text':

['ss', 'mm', 'tt', 'xx', 'tt'].

We can translate any list comprehension into equivalent code that uses more familiar Python looping/if/list appending features.

```
x = []                      # start with an empty list
for var in iterable:        # iterate through iterable
   if p(var):               # if var is acceptable?
       x.append(f(var))     # add f(var) next in the list
```

But often using a comprehension (in the right places: where you want to create from scratch some list, tuple, set or dict) is simpler. Not all lists that we build can be written as simple comprehensions, but the ones that can are often very simple to write, read, and understand when written as comprehensions. They tend to be more efficient too.

What comprehensions aren't good for is putting information into a data structure and then mutating/changing it during the execution of the comprehension; for that job you need code more like the for loop above. So when deciding whether or not to use a comprehension, ask youself if you can specify each value in the data structure once, without changing it (as was done above, using comprehensions). Or try to write the code as a comprehension first; if you fail, then try to write it using more complicated statements in Python.

Note that we can add-to (mutate) lists, sets, and dicts, but not tuples. For tuples we would have to write this code with x = () at the top and x = x + (var,) in the middle: which builds an entirely new tuple by concatenating the old one and a one-tuple (containing only x) and then binding x to the newly constructed tuple. For large tuples, this process is very slow. Don't worry about these details, but understand that unlike lists and sets, tuples have no mutator methods: so x.append(...)/x.add(...) is not allowed.

Here is something interesting (using a set comprehension: notice {} around the comprehension.

```
x = {c for c in "I've got plenty of nothing"}   # note ' in str delimited by "
print(sorted(x))
```

It prints a set of characters (printing in a list, in sorted order, created by sorted(x)) in the string but because it is a set, each character occurs one time). So even though a few c's have the same value, only one of each appears in the set because of the semantics/meaning of sets. Note it prints as as the list

[' ', "'", 'I', 'e', 'f', 'g', 'h', 'i', 'l', 'n', 'o', 'p', 't', 'v', 'y']

because sorted takes the iterable set created and produces a sorted list as a result.

If we used a list comprehension instead of a set comprehension, the result would be much longer because, for example, the character 't' would occur 3 times in a list (but occurs only once in a set).

Note the following: binding values to for loop variables inside of comprehensions does not affect the same variable name outside the scope of the comprehension. So when running

```
x = 'ABC'                       x = 'ABC'
for x in range(1,5):            y = [x for x in range(1,5)]
   pass                         print(x)
print(x)
```

the left code prints 4 but the right code prints ABC. In the right code, the x outside the comprehension is considered a different variable than the x inside the comprehension. The reason is similar to why

```
x = 'ABC'
def f():
  x = 1
f()
```

```
print(x)
```

also prints ABC after the function call. So, a comprehension (like a function) creates its own scope for the comprehension's index variable. A for loop does not create its own scope.

Dict Comprehensions:

The form for dict comprehensions is similar, here k and v are functions (or expressions) using var. Notice the {} on the outside and the : on the inside, separating the key from the value. That is how Python knows the comprehension is a dict not a set.

```
{k(var,...) : v(var,...) for var in iterable if p(var,...)}
```

So,

```
x = {k : len(k) for k in ['one', 'two', 'three', 'four', 'five']}
print(x)
```

prints a dictionary that stores keys that are these five words whose associated values are the lengths of these words. Because dicts aren't ordered, it could print as {'four': 4, 'three': 5, 'one': 3, 'five': 4, 'two': 3}

Finally, we can write a nested comprehension, although they are harder to understand than simple comprehensions.

```
x = {c for word in ['i', 'love', 'new', 'york'] for c in word if c not in 'aeiou'}
print(x)
```

It says to collect c's, by looking in each word in the list, and looking at each character c in each word: so the c collected at the beginning is the c being iterated over in the second part of the comprehension (for c in word...).

It prints a set of each different letter that is not a vowel, in each word in the list. I could produce this same result by rewriting the outer part of the comprehension as a loop, but leaving the inner one as a comprehension (union merges two sets: does a bunch of adds).

```
x = set()                    # empty set: cannot use {} which is an empty dict
for word in ['i', 'love', 'new', 'york']:
    x = x.union( {c for c in word if c not in 'aeiou'} )
print(x)
```

or write it with no comprehensions at all

```
x = set()
for word in ['i', 'love', 'new', 'york']:
    for c in word:
        if c not in 'aeiou':
            x.add(c)
print(x)
```

So which of these is the most comprehendable: the pure comprehension, the hybrid loop/comprehension, or the pure nested loops? What is important is that we know how all three work, can write each correctly in any of these ways, and then we can decide afterwords which way we want the code to appear. As we program more, our preferences might change. I'd probably prefer the first one (because I've seen lots of double comprehensions), but the middle one is also reasonable.

What do you think the following nested (in a different way) comprehension produces?

```
x = {word : {c for c in word} for word in ['i', 'love', 'new', 'york']}
```

Check your answer by executing this code in Python and printing x.

Finally, here is a version of myprint (written above in the section on the

binding of arguments and parameters) that uses a combination of the .join function and a comprehension to create the string to print simply.

The .join function (discussed more in depth below) joins all the string values in an iterable into a string using the prefix operator as a separator:
'--'.join(['My', 'dog', 'has', 'fleas'])
returns the string 'My--Dog--has--fleas'

```
def myprint(*args,sep=' ',end='\n'):
    s = sep.join(str(x) for x in args)+end       # create string to print
    print(s,end='')                              # print the string
```

WARNING: Once students learn about comprehensions, sometimes they go a bit overboard as they learn about/use this feature. Here are some warning signs: When writing a comprehension, you should (1) use the result produced in a later part of the computation and (2) typically not mutate anything in the comprehension. If the purpose of your computation is to mutate something, don't use a comprehension. Over time you will develop good instincts for when to use comprehensions.

Tuple Comprehensions are special:

The result of a tuple comprehension is special. You might expect it to produce a tuple, but what it does is produce a special "generator" object that we can iterate over. We will discuss generators in detail later in the quarter, so for now we will examine just some simple examples. Given the code

```
x = (i for i in 'abc')  # tuple comprehension
print(x)
```

You might expect this to print as ('a', 'b', 'c') but it prints as
<generator object <genexpr> at 0x02AAD710>

The result of a tuple comprehension is not a tuple: it is actually a generator. The only thing that you need to know now about a generator is just that you can iterate over it, but ONLY ONCE. So, given the code

```
x = (i for i in 'abc')
for i in x:
    print(i)
for i in x:
    print(i)
```

it prints
a
b
c

Yes, it prints a, b, c and just prints it once: after the first loop finishes, the generator is exhausted so the second loop prints no more values. We will spend a whole lecture studying the details of generators later in the quarter.

Specifically, if x is defined as above, we cannot call len(x) or x[1]: it is not a tuple, it is a generator. All we can do it iterate over x.

Recall our discussion of changing any iterable into a list, tuple, or set by iterating over it; we can iterate over a tuple comprehension. So if we wrote t = tuple(x) or t = tuple((i for i in 'abc')), then print(t) would print ('a', 'b', 'c'). In fact, we could even write t = tuple(i for i in 'abc') because (1) by default, comprehensions are tuple comprehensions; (2) in a function call with exactly one argument, we can omit the () specifying a tuple comprehension (but with multiple arguments we must supply the parentheses).

Of course, we could also write things like :
l = list(i for i in 'abc')
s = set (i for i in 'abc')

but these are equivalent to writing the standard comprehensions more simply

```
(and efficiently) as:
l = [i for i in 'abc']
s = {i for i in 'abc'}

--------------------------------------------------------------------------------

Nine Important/Useful Functions: split/join, any/all, sum/min/max, zip/enumerate

The split/join methods

Both split and join are methods in the str class: if s is some string, we call
them by writing s.split(...) or s.join(....)

The split method also takes one str argument as .... and the result it returns
is a  list of of str. For example 'ab;c;ef;;jk'.split(';') returns the list of
str

['ab', 'c', 'ef', '', 'jk']

It uses the argument str ';' to split up the string (the one prefixing the
.split call), into slices that come before or after every ';'.

Note that there is an empty string between 'ef' and 'j' because two adjacent
semi-colons appear between them in the string. If we wanted to filter out such
empty strings, we can easily do so by embedding the call to split inside a
comprehension: writing [s for s in 'ab;c;ef;;jk'.split(';') if s != ''] 
produces the list ['ab', 'c', 'ef', 'jk'].

Because the prefix and regular arguments are both strings, students sometime
reverse these two operands: what does ';'.split('ab;c;ef;;jk') produce and why?

The split method is very useful to call after reading lines of text from a file,
to parse (split) these lines into their important constituent information.
You will use split in all 5 parts of Programming Assignment #1.

-----

The join method also takes one iterable argument (it must produce str values)
as ....; the result it returns is a str. For example
';'.join(['ab', 'c', 'ef', '', 'jk']) returns the str

'ab;c;ef;;jk'

It merges all the strings produced by iterating over its argument into one big
string, with all the strings produced by iterating over its argument
concatenated together and separated from each other by the ';' string.

So, split and join are opposites. Unfortunately, the splitting/joining string
';' appears as the argument inside the () in split, but it appears as the prefix
argument before the call in join. This inconsistency can be confusing.

--------------------------------------------------------------------------------

The all/any functions (and their use with tuple comprehensions)

The "all" function takes one iterable argument (and returns a bool value): it
returns True if ALL the bool values produced by the iterable are True; it can
stop examining values and return a False result when the first False is produced
(ultimately, if no False is produced it returns True).

The "any" function takes one iterable argument (and returns a bool value): it
returns True if ANY the bool values produced by the iterable are True; it can
stop examining values and return a True result when the first True is produced
(ultimately, if no True is produced it returns False).

These functions can be used nicely with tuple comprehensions. For example, if
we have a list l of numbers, and we want to know whether all these numbers are
prime, we can call
```

```
  all( predicate.is_prime(x) for x in 1 )
```

which is the same as calling

```
  all( (predicate.is_prime(x) for x in 1) )
```

and similar (but more time/space-efficient) than calling

```
  all( [predicate.is_prime(x) for x in 1] )
```

The list comprehension computes the entire list of boolean values and then "all" iterates over this list. When "all" iterates over a tuple comprehension, the tuple comprehension computes values one at a time and "all" checks each: if one is False, the tuple comprehension returns False immediately and does not have to compute any further values in the tuple comprehension.

-----Efficiency Interlude
The tupel comprehension version can be much more efficient, if a False value is followed by a huge number of other values.

For example, calling

```
all( predicate.is_prime(x) for x in range(2,1000000) )
```

immediately returns False because it checks whetehr 2 is prime (True), 3 is prime (True), 4 isn't prime (False: so all finishes executing, returning False).

But calling

```
all( [predicate.is_prime(x) for x in range(2,1000000)] )
```

takes a very long time to return False. It first determines whether all values from 2 to 1,000,000 are prime (putting all these boolean values in the list comprehension), and then looks at the first (True), the second (True), the third (False: so all finishes executing, returning False).

This illustrates the very important difference between tuple comprehension and other kinds of comprehensions.
-----End: Efficiency Interlude

Likewise for the "any" function, which produces True the first time it examines a True value.

Here is how we can write these functions, which search for a False or a True respectively:

```
def all(iterable):
    for v in iterable:
        if v == False
            return False  # something was False; return False immediately
    return True           # nothing   was False; return True after loop

def any(iterable):
    for v in iterable:
        if v == True
            return True   # something was True; return True immediately
    return False          # nothing   was True; return False after loop
```

Read these functions. What does each produce if the iterable produces no values? Why is that reasonable?

The sum/max/min functions (and their use with tuple comprehensions)

The simple versions of the sum function takes one iterable argument. The sum function requires that the iterable produce numeric values that can be added. It returns the sum of all the values produced by the iterable; if the iterable argument produces no values, sum returns 0. Actually, we can supply a second

argument to the sum function; in this case, that value will be returned in the special case when the iterable produces no values; if the iterable does produce any values, the sum will be the actual sum plus this argument. We can think of sum as defined by

```
def sum(values, init_sum=0):
    result = init_sum
    for v in values:
        result += v
    return result
```

The simple versions of the max and min functions also each take one iterable argument.

The min/max functions require that the iterable produce values that can be compared with each other; so calling min([2,1,3]) returns 1; and calling min(['b','a','c']) returns 'a'; but calling min([2,'a',3]) raises a TypeError exception because Python cannot compare an integer to a string. These functions return the minimum/maximum value produced by their iterable argument; if the iterable produces no values (e.g., min([]), min and max each raise a ValueError exception, because there is no minimum/maximum value that it can compute/return.

There are two more interesting properties to learn about the min and max functions.

First, we can also call min/max specifying any number of arguments or if one argument, it must be iterable: so calling min([1,2,3,4]) -using a tuple which it iterable- produces the same result as calling min(1,2,3,4) -using 4 arguments.

We can also specify a named argument in min/max: a key function just like the key function used in sort/sorted. The min/max functions return the smallest/largest value in its argument(s), but if the key function is supplied, it compares two values by calling the key function on each, not by directly comparing these two value.

For example, min('abcd','xyz') returns 'abcd', because 'abcd' < 'xyz'. But if we instead wrote min('abcd','xyz',key = (lambda x : len(x))) it would return 'xyz' because len('xyz') < len('abcd'): that is, it compares the key function applied to all values in the iterable to determine which value to return.

Note that min('abcd','wxyz',key = (lambda x : len(x))) will return 'abcd' because it is the first value that has the smallest result when the key function is called: the key function returns 4 for both values. We can think of the min function operating on an iterable to be written as follows (although we will learn Python features later on that simplifies the actual definition of this function)

```
def min(*args,key = (lambda x : x) ): # default key is the identity function
    if len(args) == 0:
        raise TypeError('min: expected >=1 arguments, got 0')
    if len(args) == 1:     # Assume that if min has just one argument
        args = args[0]     #   it's iterable, so take the max over its values

    answer = None
    for v in args:
        key_of_v = key(v)
        if answer == None or key_of_v < key_of_answer:
            answer = v
            key_of_answer = key_of_v

    return answer
```

Calling min(('abc','def','gh','ij'),key = (lambda x : len(x))) returns 'gh'. Note that the default value for the key function is a lambda that returns the value it is passed: this is called the identity function/lambda.

The zip and enumerate functions

Zip:

There is a very interesting function called zip that takes an arbitrary number of iterable arguments and zips/interleaves them together (like a zipper does for the teeth on each side). Let's start by looking at just the two argument version of zip.

What zip actually produces is a generator -the ability to get the results of zipping- not the result itself. See the discussion above about how tuple comprehensions produce generators.

So to "get the result itself" we should use a for loop or constructor (as shown in most of the examples below) to iterate over the generator result of calling zip. The following code

```
z = zip( 'abc', (1, 2, 3) )           # String and tuple are iterator arguments
print('z:', z, 'list of z:', list(z))
```

prints

```
z: <zip object at 0x02A4D990> list of z: [('a', 1), ('b', 2), ('c', 3)]
```

Here, z refers to a zip generator object; the result of using z in the list constructor is [('a', 1), ('b', 2), ('c', 3)] which zips/interleaves the values from the first iterable and the values from the second:
[(first from first,first from second),(second from first,second from second),
 (third from first,third from second)]

What happens when the iterables are of different lengths? Try it.

```
z = zip( 'abc', (1, 2) )  # String and tuple for iterables
print(list(z))            # prints [('a', 1), ('b', 2)]
```

So when one iterable runs out of values to produce, the process stops. Here is a more complex example with three iterable parameters of all different sizes. Can you predict the result it prints: do so, and only then run the code.

```
z = zip( 'abcde', (1, 2, 3), ['1st', '2nd', '3rd', '4th'] )
print(list(z))
```

which prints

```
[('a', 1, '1st'), ('b', 2, '2nd'), ('c', 3, '3rd')]
```

Of course, this generalizes for any number of arguments, interleaving them all (from first to last) until any iterable runs out of values. So the number of values in the result is the minimum of the number of values of the argument iterables.

Note one very useful way to use zip: suppose we want to iterate over values in two iterables simultaneously, i1 and i2, operating on the first pair of values in each, the second pair of values in each, etc. We can use zip to do this by writing:

```
for v1,v2 in zip(i1,i2):
    process v1 and v2: the next pair of values in each
```

So

```
for v1,v2 in zip ( ('a','b','c'), (1,2,3) ):
    print(v1,v2)
```

prints

```
a 1
b 2
c 3
```

Using zip, we can write a small function that computes the equivalent of the < (less than) operator for strings in Python (see the discussion above about the meaning of < for strings).

```
def less_than(s1 : str, s2 : str)-> bool:
    for c1,c2 in zip(s1,s2):   # examine 1st, 2nd, ... characters of each string
        if c1 != c2:           # if current characters are different
            return c1 < c2     #   compute result by comparing characters

    # if all character from the shorter are the same (as in 'a' and 'ab')
    # return a result based on the length of the strings
    return len(s1)<len(s2)
```

This precisely caputures in Python code our prose discussion of the meaning of comparing strings.

Enumerate:

Finally, this is a convenient time to toss in another important function: enumerate. It also also produces a generator as a result, but has just one iterable argument, and an optional 2nd argument that is a starting number. It produces tuples whose first values are numbers (starting at the value of the second parameter; omit a 2nd parameter and unsurprisingly, the starting number is 0) and whose second values are the values in the iterable. So, if we write

```
e = enumerate(['a','b','c','d'], 5)
print(list(e))
```

it prints [(5, 'a'), (6, 'b'), (7, 'c'), (8, 'd')]

Given l = ['a','b','c','d','e'] we could write the following code

```
for i in range(len(l)):
    print(i+1,l[i])
```

(which prints 1 a, 2 b, 3 c, 4 d, and 5 e on separate lines) more simply by using enumerate (notice the use of parallel/tuple assignment for i,x):

```
for i,x in enumerate(l,1):
     print(i,x)
```

Another nice example illustrating enumerate is reading a file a line at a time, and processing the line number and the line contents.

Instead of writing

```
line_number = 1
for line in open("file-name"):
    .... process line_number and line
    line_number += 1
```

we can write just

```
for line_number, line in enumerate(open("file-name"),1):
    .... process line_number and line
```

You might ask now, why do these things (tuple comprehension, zip, enumerate) produce generators and not just tuples or lists? That is an excellent question. It goes right along with the excellent question why does sorted(...) produce a list and not a generator? We will discuss these issues later in the quarter. The generator question mostly has to do with space efficiency when iterating over very many values. The sorted question has to do with why there is no way to do this operation in a space efficient way.

**kargs for dictionary of not-matched named arguments in function calls

Recall the use of *args in a function parameter list. It combines into one

tuple a sequence of positional arugments. We can also write the symbol **kargs (we can write ** and any word, but kargs or kwargs are the standard ones). If we use it to specify this kind of parameter, it must occur as the last parameter. kargs stands for keyword arguments. Basically, if Python has any keywords arguments that do not match keyword parameters (see the large discussion of argument/paramteter binding above, which includes *args but doesn't include **kargs) they are all put in a dictionary that is stored in the last parameter named kargs.

So, imagine we define the following function

```
def f(a,b,**kargs):
  print(a,b,kargs)
```

and call it by

```
f(c=3,a=1,b=2,d=4)
```

it prints: 1 2 {'c': 3, 'd': 4}

Without **kargs, using the rules specified before, Python would report a TypeError exception (by rule M5(d)), because there are no parameter named c or d.

By the new rules, Python finds two named arguments (c=3 and d=4) whose names did not appear as parameter names in the function header of f (which specifies only a and b, and of course the special **kargs), so while Python directly binds a to 1 and b to 2 (the parameter names specified in the function header, matched to similarly named arguments) it creates a dictionary with all the other named arguments: {'c': 3, 'd': 4} and binds kargs to that dict.

The same result would be printed for the call

```
f(1,2,d=4,c=3)
```

We will use **kargs to understand a special use of of dict constructors (below). We will also use **kargs (and learn something new in the process) when discussing how to (1) call methods in decorators and (2) call overridden methods using inheritance much later in the quarter.

Note that to write a perfectly general function that is called with any kinds of arguments we can write

```
def g(*args,**kargs):
  print(args,kargs)
```

Now, any legal call of g (one with any legal combination of positional and named arguments) will populate the *args and **kargs structures appropriately.

Calling

```
g(1,2,c=3,d=4)     prints (1, 2) {'c': 3, 'd': 4}
g(a=1,b=2,c=3,d=4) prints () {'c': 3, 'b': 2, 'a': 1, 'd': 4}
```

Generally, when a parameter name is prefixed by * and ** in a function header, we omit the prefix when we use the parameter name inside the function. But there are times where use use the * and ** prefixes.

```
def h(a,b,c,d):
    print(a,b,c,d)

def i(*args,**kargs):
  h(*args,**kargs)
```

The arguments *args and **kargs in the call of h expand the args tuple to be positional arguments and the **kargs dictionary to be named arguments

Calling

```
i(1,2,c=3,d=4)     prints 1 2 3 4: it calls h(1,2,c=3,d=4)
i(a=1,b=2,c=3,d=4) prints 1 2 3 4: it calls h(c=3,b=2,a=1,d=4)

Summarizing:

1) Writing * and ** when specifying parameters makes those parameters names
   bind to a tuple/dict respectively.

2) Using the parameter names by themselves in the function is equivalent to
    using the tuple/dict respectively.

3) Using * and ** followed by the parameter name as ARGUMENTS IN FUNCTION
   CALLS expands all the values in the tuple/dict respectively to represent
   all the arguments.

Experiment with *args/**kargs as parameters of functions and args/kargs and
*args/**kargs as arguments to other function calls.

-----Interlude
We have seen that to call sorted with a lambda we must write something like

  sorted(iterable, key = (lambda ...))

One might ask, is the key paraemeter to sorted the second one? Could we write

  sorted(iterable, (labmda ...))

to accomplish the same call? The answer is "no" we cannot. If we want to use
a lambda, we must write a named argument as  key = (lambda ...). Similarly, we
must write reverse = ....

How do you think the header of the sorted function is written?

--------------------------------------------------------------------------------

Lists, Tuples, Sets, Dictionaries (frozenset and defaultdict)

You need to have pretty good grasp of these important data types, meaning how to
construct them and the common methods/operations we can call on them. Really
you should get familiar with reading the online documentation for all these
data types (see the Python Library Reference link on the homepage for the
course).

4. 6: Sequence Types includes Lists (mutable) and Tuples (immutable)
4. 9: Set Types includes set (mutable) and frozenset (immutable)
4.10: Mapping Types includes dict and defaultdict (both mutable)

Here is a very short/condensed summary of these operations. Experiment with
them in Eclipse.

-----

4.6: Sequence Types includes Lists (mutable) and Tuples (immutable)
These sequence operations (operators and functions) are defined in 4.6.1
   x in s, x not in s, s + t, s * n, s[i], s[i:j], s[i:j:k], len(s), min(s),
   max(s), s.index(x[, i[, j]]), s.count(x)

Mutable sequence allow the following operations, defined in 4.6.3
  s[i] = x , s[i:j] = t, del s[i], s[i:j:k] = t, del s[i:j:k], s.append(x)
  s.clear(), s.copy(), s.extend(t), s.insert(i, x), s.pop(), s.pop(i),
  s.remove(x), s.reverse()

  It also discusses list/tuple constructors and sort for list.

  note that the append method is especially important for building up sequences
  like lists. Also to return and remove a value from a sequence, we call
    x = s.pop(i)
  is equivalent to
    x = s[i]
```

```
    del s[i]
  and calling s.pop() uses the value len(s)-1 (the last index in the sequence)

-----

4. 9: Set Types includes set (mutable) and frozenset (immutable)
These set (operators and functions) are defined in 4.6.1.9
  len(s), x in s, x not in s, isdisjoint(other), issubset(other), set <= other,
  set < other, issuperset(other), set >= other, set > other, union(other, ...),
  intersection(other, ...), difference(other, ...), symmetric_difference(other),
  copy; also the operators | (for union), & (for intersection), - (for
  difference), and ^ (for symmetic difference)

Sets, which are mutable, allow the following operations
  update(other, ...), intersection_update(other, ...),
  difference_update(other, ...), symmetric_difference_update(other), add(elem),
  remove(elem), discard(elem), pop(), clear(); also the operators |= (union
  update), &= (intersection update), -= (difference update), ^= (symmetric
  difference update)

-----

4.10: Mapping Types includes dict and defaultdict (both mutable)
These dict (operators and functions) are defined in 4.10
  d[key] = value , del d[key], key in d, key not in d, iter(d), clear(), copy(),
  fromkeys(seq[, value]), get(key[, default]), items(), keys(),
  pop(key[, default]), popitem(), setdefault(key[, default]), update([other]),
  values()

Important Notes on dicts:
  d[k] returns the value associated with a key (raises exception if k not in d)

  d.get(k,default) returns d[k] if k in d; returns default if k not in d
    it is equivalent to the conditional expression (d[k] if k in d else default)

  d.setdefault(k,default) returns d[k] if k in d; if k not in d it
    (a) sets d[k] = default
    (b) returns d[k]
  writing d.setdefault(k,default) is equivalent to (but more efficient than)
  writing
      if k in d:
          return d[k]
      else
          d[k] = default
          return d[k]

There is a type called defaultdict (see 8.3.4) whose constructor generally
takes an argument that is a reference to any object that CAN BE CALLED WITH NO
ARGUMENTS. Very frequently we use a NAME OF A CLASS that when called will
CONSTRUCT A NEW VALUE: if the argument is int, it will call int() producing the
value 0; if the argument is list, it will call list() producing an empty list;
if the argument is set, it will call set() producing an empty set; etc.

Whenever a key is accessed for the first time (i.e., that key is accessed but
not already associated with a value in the dictionary) in a defaultdictionary,
it will associate that key with the value created by calling the reference to
the object supplied to the constructor.

Here is an example of program first written with a dict, and simplified later
by using a defaultdict.

letters = ['a', 'x', 'b', 'x', 'f', 'a', 'x']
freq_dict = dict()             # could use = {}
for l in letters:
    if l not in freq_dict:     # must check l in freq_dict before freq_dict[l]
        freq_dict[l] = 1       # if not there, put with frequency of 1
    else:
        freq_dict[l] += 1      # otherwise there, increment frequency
print(freq_dict)
```

This would print the following dict: {'b': 1, 'x': 3, 'f': 1, 'a': 2}

As each letter in the loop is processed, it is associated with 1 (if not already in the dict) or it is in the dict, and its associated value is incremented by 1.

We could solve this a bit more easily with a defaultdict.

```
from collections import defaultdict # in same module as namedtuple
letters = ['a', 'x', 'b', 'x', 'f', 'a', 'x']
freq_dict = defaultdict(int)   # int not int(); but int() returns 0 when called
for l in letters:
    freq_dict[l] += 1          # in dict, exception raised if l not in d, but
print(freq_dict)               #   defaultdict calls int() putting 0 there first
```

As each letter in the loop is processed, its associated key is looked up: if the key is absent, it it placed in the dict assocatiated with int()/0; then then the associated value (possibly the one just put in the dict) is incremented by 1.

The dict code below is equivalent to how the defaultdict code above works.

```
letters = ['a', 'x', 'b', 'x', 'f', 'a', 'x']
freq_dict = dict()             # could use = {}
for l in letters:
    if l not in freq_dict:     # must check l in freq_dict before freq_dict[l]
        freq_dict[l] = int()   # int() constructor returns 0; could write 0 here
    freq_dict[l] += 1          # l is guaranteed in freq_dict, either because
print(freq_dict)               #   it was there originally, or just put there
```

Another way to do the same thing (but also a bit longer and less efficient) uses the setdefault method (listed above)

```
letters = ['a', 'x', 'b', 'x', 'f', 'a', 'x']
freq_dict = dict()             # note dict only
for l in letters:
    freq_dict[l] = freq_dict.setdefault(l,0) + 1
print(freq_dict)
```

Here we evaluate the right side of the = first; if l is already in the dict, its associated value is returned; if not, l is put in the map with an associated value of 0, then this assocated value is returned (and then incremented, and stored back into freq_dict[l] replacing the 0 just put there).

You should achieve a good understanding of why each of these four scripts work and why they all produce equivalent results.

Often we use defaultdicts with list instead of int: just as int() produces the object 0 associated with a new key, list() creates an empty list associated with a new key (and later we add things to that list); likewise we can use defaultdicts with set to get an empty set with a new key.

So, if we wrote x = defaultdict(list) and then immediately wrote x['bob'].append(1) then x['bob'] is associated with the list [1] (the empty list, with 1 appended to it). If we then wrote x['bob'].append(2), then x['bob'] is associated with the list [1, 2]. So, we can always append to the value associated with a key, because it will use an empty list to start the process if there is nothing associated with a key.

How could we specify a defaultdict that initializes keys with the value 5? Translation: in what simple way can we pass to defaultdict an arguement that is a function of no parameters, which returns the value 5. The solution can lead to all sorts of interesting variants, say for a defaultdict whose unknown keys are automatically associated with a dict (or another defaultdict).

Note that the ==/!= operators for dictionaries work correct between dicts and defaultdicts. They are considered equal so long as they have the equal keys, and each key is associated with an equal value.

Just as with comprehensions, what is important is that we know how things like defaultdicts and dicts (with the setdefault method) work, so that we can correctly write code in any of these ways. We can decide afterwords which way we want the code to appear. As we program more, our preferences might change. I have found defaultdicts are mostly what I to use to simplify my code, but every so often I must use a regular dict.

Later in the quarter we will use inheritance to show how to write the defaultdict class simply, by extending the dict class.

Printing Dictionaries in Order: An example of using comprehensions and sorted

In this section we will combine our knowledge about the sorted function, comprehensions, and iteratring over dictionaries to examine how we can print (or generally process) dictionaries in arbitrary orders.

Generally, all of our code will be of the form

```
  for index(es) in sorted( iterable, key = (lambda x : ....) ):
     print( index(es) )
```

In these examples, we will use the simple dictionary

```
   d = {'x': 3, 'b': 1, 'a': 2, 'f': 1}
```

In each example, we will discuss the relationships among index(es), iterable, and the (lambda x : ....) in the function bound to the key parameter.

1) In the first example, we will print the dictionary keys in increasing alphabetical order, and their associated values, by iterating over d.items().

```
for k,v in sorted( d.items(), key = (lambda item : item[0]) ):
   print(k,'->',v)
```

which prints as

```
a -> 2
b -> 1
f -> 1
x -> 3
```

Here, iterable is d.items(), which produces 2-tuples storing each key and its associated value, for every item in the dictionary; the item in lambda item is also a key/value 2-tuple (specifying here to sort by item[0], the key part in the 2-tuple); finally, the list returned by sorted also contains key/value 2-tuples, which are unpacked into k and v and printed.

We can solve this same problem by iterating over just the keys in d as well.

```
for k in sorted( d.keys(), key = (lambda k : k) ):
   print(k,'->',d[k])
```

Here, iterable is d.keys() which produces strings storing each key in the dictionary; the k in lambda k is also a key/str value (specifying here to sort by k, the key itself: I could have omitted this identity lambda); finally, the list returned by sorted also contains key/str value, which are stored into k and printed along with d[k].

This code is equivalent to the following, since d.keys() is the same as just d, and lambda x : x is the default for the key parameter.

```
for k in sorted( d ):
   print(k,'->',d[k])
```

2) In the second example, we will print the dictionary keys and their associated values, in increasing order of the values, by iterating over d.items().

```
for k,v in sorted( d.items(), key = (lambda item : item[1]) ):
   print(k,'->',v)
```

which prints as

```
b -> 1
f -> 1
a -> 2
x -> 3
```

Here, iterable is d.items(), which produces 2-tuples storing each key and its associated value, for every item in the dictionary; the item in lambda item is also a key/value 2-tuple (specifying here to sort by item[1], the value part in the 2-tuple); also, the list returned by sorted also contains key/value 2-tuples, which are unpacked into k and v and printed. Finally, by this lambda either b or f might be printed first, because they have the same value associated with them.

We can solve this same problem by iterating over just the keys in d as well.

```
for k in sorted( d.keys(), key = (lambda k : d[k]) ):
   print(k,'->',d[k])
```

Here, iterable is d.keys() which produces strings storing each key in the dictionary; the k in lambda k is also a key/str value (specifying here to sort by d[k], the value associated with the key); finally, the list returned by sorted also contains key/str values, which are stored into k and printed along with d[k].

This code is equivalent to the following, since d.keys() is the same a d; the lambda is still needed here because it is not the identity lambda.

```
for k in sorted( d, key = (lambda k : d[k]) ):
   print(k,'->',d[k])
```

In both cases, Python uses the LEGB rule: d is not a parameter to the lambda, but it might be defined in the Enclosing or Global scope, so it can be used in the lambda.

3) In the third example, we will compute a list that contains all the keys in a dictionary, sorted by their associated values (in decreasing order of the values), by iterating over d.keys() (really just d, to simplify the code)

We compute

```
ks = sorted(d, key = (lambda k : d[k]), reverse = True)
```

ks stores ['x', 'a', 'b', 'f'].

We could also compute this list less elegantly using d.items().; I say less elegantly, because we need a comprehension to "throw away" the value part of each item returned by sorted.

```
ks = [k for k,v in sorted(d.items(), key=(lambda item : item[1]), reverse=True)]
```

4) Finally, in the fourth example, we will compute a list that contains all the 2-tuples in the items of d, but the tuples are reversed (values before keys) and sorted in increasing alphabetical order by keys.

We compute this list of 2-tuple in the following two ways

```
vks = [ (v,k) for k,v in sorted( d.items(), key = (lambda item : item[0]) ) ]
vks = sorted( ((v,k) for k,v in d.items()), key = (lambda item : item[1]) )
```

vks stores [(2, 'a'), (1, 'b'), (1, 'f'), (3, 'x')]

The top computation first creates a sorted version of d by keys, and then uses a

comprehension to create tuples that reverse the keys/values; the bottom computation first uses a comprehension to create a list of reversed keys/values, and then uses sorted to sort these reversed 2-tuples by the keys that are now in the second part of each tuple.

Using _ as a variable Name:

If you write

```
  ks = [k for k,v in list_of-2-tuples...]
```

Eclipse will likely give you warning about "variable v is not used". Since there must be variable to store the second tuple index, but that variable does not get used, we can write it as just _ (a legal Python variable name)

```
  ks = [k for k,_ in list_of-2-tuples...]
```

In this case, Eclipse does not indicate a warning. Naming a variable _ implies you aren't going to to anything useful with it. Although, we can write

```
for _ in [1,2,3]:
    print(_)
```

and it will print 1, 2, and 3 (on their own lines). By using _ Eclipse doesn't expect this name to be used.

Exceptions: example from prompt_for_int

We do two things with exceptions in Python: we raise them (with the raise statement) and we handle them (with the try/except statement). Exceptions were not in early programming languages, but were introduced big time in the Ada language in the early 1980s, and have been parts of most new languages since then.

A function raises an exception if it cannot do the job it is being asked to do. Rather than fail silently, possibly producing a bogus answer that gets used to compute a bogus result, it is better that the computation announces a problem occurred and if no way is known to recover from it (see handling exceptions below) the computation halts with an error message to the user.

For example, if your program has a list l and you write l[5] but l has nothing stored at index 5 (its length is smaller), Python raises the IndexError exception. If you haven't planned for this possibility, and told Python how to handle the exception and continue the calculation, then the program just terminates and Python prints:

IndexError: list index out of range

Why doesn't it print the index value 5 and the lenth of list l?. I don't know. That certainly seems like important and useful information, and Python knows those two values. I try to make my exception messages include any information useful to the programmer.

There are lots of ways to handle exceptions. Here is a drastically simplified example from my prompt class (this isn't the real code, but a simplification for looking at a good use of exception handling). But it does a good job of introducing the try/except control form which tries to execute a block of code and handles any exceptions it raises.

If we write a try/except and specify the name of no exception, it will handle any exception. The name Exception is the name of the most generic exception.

We can write a try/except statement with many excepts, each one specifying a specific exception to handle, and what to do when that exception is raised. In fact, int('x') raises the ValueError exception, so I use ValueError in the except clause below to be specific, and not accidentally handle any other kind

of exception.

```
def prompt_for_int(prompt_text):
    while True:
        try:
            response = input(prompt_text+': ')    # response is used in except
            answer = int(response)
            return answer
        except ValueError:
            print('  You bozo, you entered "',response,'"',sep='')
            print('  That is not a legal int')

print(prompt_for_int('Enter a positive number'))
```

So, this is an "infinite" while loop, but there is a return statement at the bottom of the try-block; if it gets executed, it returns from the function, thus terminating the loop. The loop body is a try/except; the body of the try/except

1) prompts the user to enter a string on the console (this cannot fail)

2) calls int(response) on the user's input (which can raise the ValueError exception, if the user types characters that cannot be interpreted as an integer)

3) if that exception is not raised, the return statement returns an int object representing the integer the user typed as a string

But if the exception is raised, it is handled by the except clause, which prints some information. Now the try/except is finished, but it is in an infinite loop, so it goes around the loop again, reprompting the user (over and over until the user enters a legal int).

Actually, the body of try could be simplified (with the same behavior) to just

```
  response = input(prompt_text+': ')    # response is used in except
  return int(response)
```

If an exception is raised while the return statement is evaluating the int function, it still gets handled in except. We CANNOT write it in one line because the name response is used in the except clause (in the first print). If this WASN'T the case, we could write just

```
  return int(input(prompt_text+': '))    # if response not used in except
```

For example, we might just say 'Illegal input' in the except, but in the example above, it actually display the string (not a legal int) that the user typed.

Finally, in Java we throw and catch exceptions (obvious opposites, instead of raise and handle) so I might sometimes use the wrong term. That is because I think more generally about programming than "in a language", and translate what I'm thinking to the terminology of the language I am using, but sometime I get it wrong.

Note that exception handling is very powerful, but should be avoided if a boolean test can easily determine whether a computation will fail. For example.

```
l = [...]
if (0 <= i < len(l))
  print(l[i])
```

is preferred to

```
try:
  print(l[i])
except IndexError:
  pass
```

Name Spaces (for objects): __dict__

Every object has a special variable named __dict__ that stores all its namespace bindings in a dictionary. During this quarter we will systematically study class names that start and end with two underscores. Writing x.a = 1 is similar to writing x.__dict__['a'] = 1; both associate a name with a value in the object. We will explore the uses of this kind of knowledge in much more depth later in the quarter.

Here is a brief illustration of the point above. Note that there is a small Python script that illustrates the point. This is often the case.

```
class C:
    def __init__(self): pass

o = C()
o.a = 1
print(o.a)           # prints 1

o.__dict__['a'] = 2
print(o.a)           # prints 2

o.__dict__['b'] = 3
print(o.a,o.b)       # prints 2 3
```

Trivial Things.

An empty dict is created by {} and empty set by set() (we can't use {} for an empty set because Python would think it is a dict). Any non-empty dicts can be distinguished from a non-empy set because a non-empty dict will always have the : character inside {} separating keys from values. Suppose you want to create a set containing the value 1: which works (and why)? {1} or set(1).

A one value tuple must be written like (1,) with that "funny" comma (we can't write just (1) because that is just the value 1, not a tuple storing just 1).

Questions:

1. Describe, in term of binding, what happens if the first statement in a module is x = 0, in terms of the module object and its namespace.

2. Assume that we have executed the statement x = 1. Describe, in terms of binding, what is the semantics of the statement x += 1.

3. What is printed in print(f(0)) if we define f as follows?

```
def f(x):
    pass
```

4. Using the kind of pictures dicussed in the Binding section above, illustrate the meaning of a module named m that contains the following statements:

```
x = 1
y = 2
z = y
```

And a module named script that contains the following statements:

```
import m
a = m.x
m.y = 10
from m import y
from m import z as b
z = y
```

```
  del m.z
  c = 10
```

The result will be two large rounded-rectangles (objects for modules m and script) which contain labeled boxes that refer to other rounded-rectangles (objects for int values), some of which are shared (referred to by multiple names)

5. Predict what the following script will produce and explain why.

```
print(print(5))
print=1
print(print)
```

6a. Write a simple function named count that returns the number of times a value is in a list; write the a simple function named indexes that returns a list of indexes in a list that a value returns. Use the appropriate kind of for loop. For example, count(5, [5,3,4,5,1,2,5]) returns 3; indexes(5, [5,3,4,5,1,2,5]) returns [0,3,6].

7. Suppose that we define the functions double, triple, and times10 (as done above). Write the function call, such that call('double',5) returns the result double(5); call('triple',5) returns the result triple(5); call('magnitude',5) returns the result magnitude(5). Hint: use eval.

8. Write a function named between that is defined with two parameters. It returns a reference to a function that has one parameter, which returns whether or not its one parameter is between (inclusive) the two arguments supplied when between is called. For example

```
college_age = between(18,22)
print( college_age(20) )
```

print True because 18 <= 20 <= 22.

9. Assume s = 'Fortunate'. Explain how Python evalues s.replace('t','c'). Sure, the result it produces is 'Forcunace', but exactly how does Python know which function to call and how to call that function with these arguments. Hint: Use the Fundamenal Equation of Object-Oriented Programming.

9.5 This is tricky: it requires a good understanding of exec, eval, and what characters really appear in strings when you type them on the console.
(1) prompt the user to enter a string that defines a function (using \n where new lines would be in the function); (2) use exec to define the function;
(3) prompt the user to enter a call to the function and print what it evaluates to. The interaction might look like:

```
Enter function definition as string: def f(n):\n    return 2*n
Enter function call as string      : f(3)
6
```

Hint: when you enter \n in a string, what characters are entered? Use the replace function to "fix" them to exec will work.

10. Assume that you have a list of tuples, where each tuple is a name and birthday: e.g., ('alex', (9, 4, 1988)) and ('mark', (11, 29, 1990)). The birthday 3-tuple, consists of a month, followed by day, followed by year. Write a for-loop using sorted and a lambda to print out the tuples in the list in order of their bithdays (just months and days) so the 'alex' tuple prints before the 'mark' tuple because September 4th comes before November 29th; all people who have the same birthday should be printed in alphabetical order. Hint write a lambda that uses a key that is a person's month value, followed by a day value, followed by a name value.

11. What does the following script print?

```
print('a','b','c',sep='.-x-.')
print('d','e','f',sep='')
print('g','j','i',sep=':',end='..')
```

```
print('j','k','l',sep='--',end='?')
print('m',end='\n\n')
print('n')
```

12. Assume s is a string. What is the value of s[-1:0:-1]? Assume l is a list. Write the simplest loop that prints all values in a list except the first and last (if the list has2 or fewer values, it should print nothing).

13. Write a single Python statement using a conditional expression that is equivalent to the following two statementso

```
t= 0
if x < 0:
    t = -1
```

14. What can we say about

```
  len of set(d.keys())   compared to len of d.keys?
  len of set(d.values()) compared to len of d.values()?
  len of set(d.items())  compared to len of d.items()?
```

15. Assume we import the copy function from the copy module. What is the difference between the result produced by list((1,2,3)) and copy((1,2,3))?

15.5 Using the kind of pictures dicussed in the Binding section above, illustrate the meaning of the following and determine what is printed.

```
  x = [ [0], [1]]
  y = list(x)
  x[0][0] = 'a'
  print(x,y)
```

16a. Assume we have a list of students (ls) and a list of their gpas (lg). Create a dictionary whose first key is the first student in ls and whose associated value is the first gpa in the lg; whose second key is the second student in ls and whose associated value is the second gpa in the lg; etc. So if ls = ['bob','carol','ted','alice'] and lg = [3.0, 3.2, 2,8, 3.6] the resulting dict might print as {'carol': 3.2, 'ted': 2, 'alice': 8, 'bob': 3.0}. Hint: use the dict constructor and the zip function.

16b. Assume that we have a list of runners in the order they finished the race. For example ['bob','carol','ted','alice']. Create a dictionary whose keys are the students and whose values are the place in which they finished. For this example the resulting dict might print as
{'carol': 2, 'ted': 3, 'alice': 4, 'bob': 1}. Hint: use a comprehension and enumerate.

16c. Assume that we have a list of values l and a function f, and we want to define a function that computes the value x in l whose f(x) is the smallest. For example, if l = [-2, -1, 0, 1, 2] and def f(x) : return abs(x+1), then calling min_v(l,f) would return -1. Hint: try to write the min_v function in one line: a returns statement calling min, on a special comprehension, and indexing.

17. The following script uses many topics that are covered in the lecture.

Write a script that reads a multiline file of words separated by spaces (no punctuation) and builds a dictionary whose keys are the words and whose values are lists of the line numbers that the words are on (with no duplicate line numbers in each list). Print all the words (in sorted order, one per line) with the list of their line numbers afterwards. For example, the 3 line file

```
to be or
not to be
that is the question
```

would print as

```
be [1, 2]
is [3]
not [2]
```

```
or [1]
question [3]
that [3]
the [3]
to [1, 2]
```

My solution was 8 lines long (including one import). It used a defaultdict, three loops, three function calls (open, enumerate, and sorted), and four method calls (rstrip, split, append).