6.189 – Notes/Homework
Session 6

Administrivia

Name:

Instructions:

1. Err..complete the questions :). Putting a serious effort into this homework is **mandatory** for receiving a passing grade in this course (note the word effort – getting them all wrong is ok, but skipping this homework is not.)
2. These notes (and homework problems) are a continuation of the notes from Exam 1.
3. Some problems may be marked **{alone}** (it'll be in red and braces.) **Work on those problems alone!**
4. When we ask for output, you DON'T have to write the spaces/newlines in.

Program Text:

```
print "X",
print "X",
```

Output:

```
XX
```

Day 4: List Basics

Notes:

This section covers the *basics* of **lists**. We'll cover them in more detail after we cover **objects** and **references**.

We've used variables to store basic information such as numbers, strings, and boolean values. We can also store more complex information. A **list** stores a sequence of elements `[x1,x2,x3,...]`. Note that order matters: `[x1,x2] != [x2,x1]`

You can store different types of variables in the same list: `[1,3,"test",True]`. You can even store lists in lists! (Note: For now, *don't* store lists in lists! You'll understand why when we talk about objects and mutability.)

Chapters 9.1-9.3 cover basic syntax on using lists. Here's a quick review:

- You can create a list using square brackets `[1,2,5,1,3,"test"]`.
- To read an element from list `some_list`, use `some_list[index]`. The index of an element is just its position, but note that the first element is considered to have a position of 0!

- The built-in function `len(x)` will find the length of any list passed in as a parameter. This works more generally with any complex data structure (like dictionaries, which we'll cover later) *as well as strings*.

We'll also be using the following syntax for adding something to the **end** of a list.

Program Text:

```
some_list = [1,2,3,4]
some_list.append(5)
print some_list
```

Output:

```
[1,2,3,4,5]
```

For now, just memorize the syntax -- you'll understand what's going on in a later section.

Problem 12 {alone}:
What is the output of the following code. **The output of each program has at most two semi-colons.**

Program Text:

```
some_list = [3,6,2,5]
i = 1
while i < 3:
  print some_list[i], ";"
  i = i + 1
print some_list[3]
```

Output:

```

```

Program Text:

```
list1 = [2,5,6,3,7,8,12]
list2 = [1,2,6,5,7,3,12]
i = 0
while i < 6:
  if list1[i] == list2[i]:
    print list1[i], ";",
  i = i + 1
print len(list1)
```

Output:

```

```

Day 6: Objects and References

Notes:

Lists are fundamentally different than everything we've seen to date. They are the first example we've seen of an **object**.

You've learned that we can use variables to store information in a table. This works for simple things like numbers, but not for complex information like a list. Instead, Python stores this information in an area of memory called the **heap**.

Understanding what the **heap** is isn't important. What is important to know, however, is that instead of storing objects in the variable table, Python stores a **pointer** or **reference** to the object.

A reference is like an arrow. The easiest way to explain this is with an example:

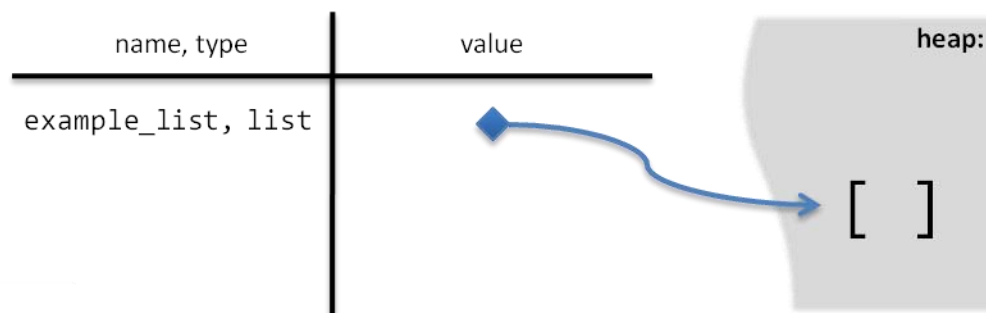
Program Text:

```
example_list = []

def append_one(my_list):
  "This is a function that adds something to the end of a list"
  #Point 2
  my_list.append(1)
  #Point 3

#Point 1
append_one(example_list)
print example_list
#Point 4
```

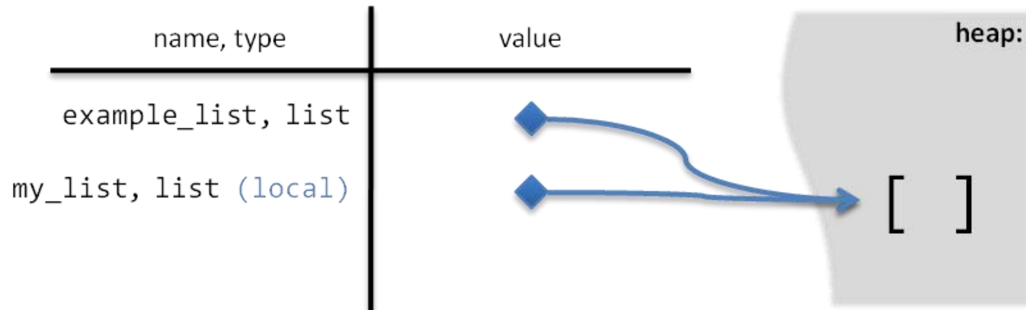
Output:

```
[1]
```

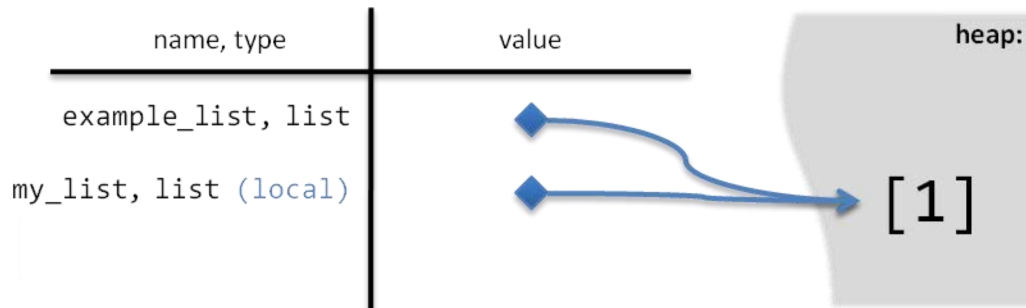
At **point 1**, the variable table looks something like the following.



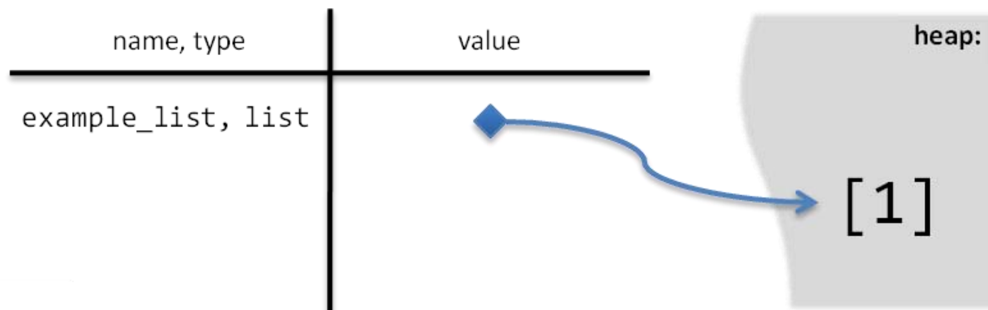
After this, we pass the location of the list (which is the value stored in the variable) to the function. Thus, at **point 2** both variables point to the same thing.



We change the list afterwards, and the table looks like the following at **point 3**.



The local variable disappears when the function ends, and at **point 4** the variable table looks like the following.



Hence the output of the code is `[1]`! This differs from problem 10 – since these variables are storing arrows, any changes we make within the function are reflected outside of it too.

This arrow is literally stored as an integer in the table – think of it as a locker number. We can find out the value of this number using the `id(x)` function.

Program Text:

```
def small_function(my_list):
  print "in function:"
  print id(my_list)
  my_list.append(1)

example_list = []
example2_list = [3,6]
print id(example_list)
print id(example2_list)

small_function(example_list)
```

Output:

```
18162752
12860152
in function:
18162752
```

Note: Your numbers may differ from mine. The *first* and *last* numbers will still be the same, though.

You probably won't need to use the id function ever. Its really useful for figuring out whether two variables are pointing to the same list or not, though.

Problem 13:

What is the output of the following code. **Use single digits for the output of id(x).**

Program Text:

```
some_list = [1,3,2]

def f1(my_list):
  new_list = [2]
  print id(my_list), "; ",
  print id(new_list), ";",

print id(some_list), "; ",
f1(some_list)
```

Output:

```

```

Program Text:

```
some_list = [1,3,2]

def f1(my_list)
  my_list.append(7)
  print id(my_list), “;”,
  return [1,4,5]

print id(some_list), “;”,
some_list.append(5)
print id(some_list), “;”,
some_list = [4,3,7]
print id(some_list), “;”,
some_list = f1(some_list)
print id(some_list)
```

Output:

Day 6: Mutability

Notes:

Now that you understand the true nature of a list, we can talk about the concept of **mutability**. Lists are an example of a mutable object. Mutability is a synonym for *changable* – in essence, we're saying that you can change a list. Be careful -- while this sounds very simple, it can get quite tricky.

For the next problem, the references example will help a lot. Remember that statements like `list1 = [1,2,5]` change the reference (what the arrow is pointing to) and statements like `list1.append(5)` change the list itself.

Problem 14:

What is the output of the following code.

Program Text:

```
list1 = [1,2,4]
list2 = list1
list2.append(3)
print list1
```

Output:

Program Text:

```
list1 = [1,2,4]
list2 = [1,2,4]
list2.append(3)
print list1
```

Output:

Program Text:

```
example_list = [1,2,4]

def f1(my_list):
  my_list.append(3)

f1(example_list)
print example_list
```

Output:

Program Text:

```
example_list = [1,2,4]

def f1(my_list):
  my_list = [1,2,4,3]

f1(example_list)
print example_list
```

Output:

Program Text:

```
list1 = [1,2,4]
list2 = [1,2,4]
list3 = list2
list2.append(3)
list2 = list1
list2.append(3)
list1.append(3)
list1 = list3
print list1, ";",
print list3
```

Output:

```

```

We'll talk about **immutable** objects later when we cover tuples and revisit strings.

Day 6: Member functions

Notes:

We saw at the append function earlier – this function had some unusual syntax. Objects have something called **member functions**. append is a member function of a list.

In essence, member functions are basically shorthand. Python could easily have made a general `append(some_list, element)` function that appends the value of `element` to the list some_`list`. Instead, we use the member function of `some_list`: we write `some_list.append(element)` to append an element to `some_list`.

In general, member functions have the syntax `object_name.function(parameters)` – the function affects or applies to the object `object_name`.

Day 4: Lists Redux

Notes:

Now let's go over lists for real. Remember what you've just learned about them.

- Lists are **mutable objects**. When you change a list somewhere, every other reference will still point to the now changed list.
- Lists have **member functions**. For lists, these functions tend to change the list itself.

Basics: You already know how to create a list and read its members. You can also use the `example_list[i]` syntax to change values, e.g. `example_list[1] = 5`. Finally, you can write `del example_list[i]` to delete the element at position `i`.

Length: As mentioned earlier, you can use the built-in function `len(x)` to find the length of a list.

Membership: The in operator can be used to determine whether an element is a member of a list or not. `3 in [3,5,7]` evaluates to True, while `6 in [3,4,1]` evaluates to false.

Analogous to !=, you can use `not in` to check if an element is NOT a member of a list. `12 not in [3,5,7]` returns True.

Warning: Never use booleans for this operator unless the list consists ONLY of booleans. Recall that the == operator can compare different types; Python considers 0 == False and any other number equal to True. Hence, `True in [0,3,5]` evaluates to True because `3 == True` is considered True by Python.

Member functions: All of the following functions *change* the list they're called from. *They have NO return values.*

- `append(element).` Appends `element` to the end of the list.
- `insert(index, element).` Inserts `element` *before* the list element at `index`.
- `reverse()`. Reverses the order of elements in the list.
- `remove(element)`. Removes the first instance of `element` from the list.

The following function(s) don't change the list. They return information about the list instead.

- `index(element).` Returns the position of the *first instance* of `element` in the list.
- `count(element).` Returns the number of times `element` appears in the list.

Don't worry too much about memorizing all these functions. Just keep in mind that they exist.

Concatenation: You can combine two lists into a new list with the concatenation operator +.

Problem 15:
Write a definition for the following shift functions. `shift` takes a list and moves the first element to the very end. `reverse_shift` does the opposite – it moves the last element to the beginning of the list. Neither function should return anything!

Program Text:

```
def shift(my_list):
  "Shifts the list [x1,x2,...,xn] to [x2,x3,...,xn,x1]."
```

Program Text:

```
def reverse_shift(my_list):
  "Shifts the list [x1,x2,...,xn] to [xn,x1,x2,...,x(n-1)]."
```

Problem 16:
Write a definition for the function `last_index`. `last_index` takes a list and an element and returns the position of the last instance of element in list, e.g. `last_index([3,4,3,7,9],3)` returns 2. Be sure that list is in its original form when your function completes! (either create your own copy of the list or reverse any changes you make.)

Program Text:

```
def last_index(my_list, element):
  "Returns the position of the last instance of element in my_list"
```

Day 4: Advanced Positioning and Slicing

Notes:
You've already learned about the notation for selecting an element `some_list[i]` by using its position. You can also use negative numbers to select an element – the last element is considered to have position `-1`.

You can do even slicker things. `some_list[start:end]` creates a list **slice** – it returns a new list with all the elements between `start` and `end - 1` from the original list. If you omit either `start` or end, it'll select all elements from the beginning / up until the end, e.g. `some_list[1:]` creates a slice including all but the first element.

You can change the values of slices the same way you change individual values, e.g. `some_list[3:5] = [1,5,4,4,7]`. Note that the new assignment doesn't have to be the same length as the list slice you cut out.

Note, however, that if you assign the slice to a *variable*, that variable points to a new list.

Program Text:

```
list1 = [1,3,5,7]
list1[1:3] = [2,4,6]
print list1
list2 = list1[1:] #list2 points to a new list
list2.append(55)
print list1
print len(list1)
list1[3:5] = [8,8,8,8]
print list1
```

Output:

```
[1, 2, 4, 6, 7]
[1, 2, 4, 6, 7]
5
[1, 2, 4, 8, 8, 8, 8]
```

`some_list[:]` creates a *duplicate* of the list. This is very useful – since most of our list techniques destructively change the list, many times, it may be a good idea to create a duplicate of the list and mess around with that instead.

Problem 17:
What is the output of the following code.

Program Text:

```
list1 = [4,6,12]
list2 = list1[:]
list1.append(4)
print list2
```

Output:

```

```

Program Text:

```
list1 = [2,5,6,3]
i = 1
while i < len(list1):
  print list1[-i], ";",
  i = i + 1
print list1[-len(list1)]
```

Output:

```

```

Program Text:

```
list1 = [1,2,3]
i = 1
while i < len(list1):
  print list1[:i], ";",
  i = i + 1
print list1
```

Output:

```

```

Day 4: Range function

Notes:

`range(x)` is a built-in function that returns a list `[0,1,...,x-1]`, e.g. `range(3)` returns `[0,1,2]`. You can also do `range(start, finish)`, which creates the list `[start, start+1, ..., finish-1]`. Its a very useful function.

Day 4: For loop

Notes:

Like the `while` loop, the `for` loop is another control flow command – a way of executing a sequence of statements multiple times. A `for` loop executes a sequence of statements once for each element in a list.

Program Text:

```
for i in [1,2,4,6]:
  print i, ";",
print 8
```

Output:

```
1;2;4;6;8
```

The `for` loop is incredibly useful. For example, you can combine the `for` loop and `range(x)` function to execute a statement a fixed number of times.

Problem 18:
Write a definition for a function that takes a list and returns the sum of all elements in that list.

Program Text:

```
def sum_elements(my_list):
  "Returns the sum of all elements in my_list"
```

Day 4: Basic Advanced Loops

Notes:
This title is not a contradiction :p. For loops can be incredibly difficult to use correctly. This section will give you a bunch of problems to give you practice in solving problems with for loops.

Problem 19:
Each of the following function definitions takes a list as a parameter and solves a specific problem. Correctly fill the blanks in the following code to solve the problems.

There is a way to solve each problem by only filling in the blanks. Don't just add extra lines to force a solution. Also, there may even be more elegant solutions that don't use all the blanks – feel free to use those too.

Program Text:

```
def all_under_6(my_list):
  """Returns True if ALL members in my_list are under 6 (False
otherwise). my_list must be list of integers."""
  under_6 = ___________
  for i in my_list:
    if __________________:
     _________________________________________________________
  return under_6
```

Program Text:

```
def no_vowels(my_list):
  """Returns True if my_list contains only consonants (no vowels).
Assume that my_list only contains lowercase letters."""
  vowels = ['a','e','i','o','u']
  contains_vowels = _________
  for i in __________________:
    if ______________________:
      ____________________________________________________________
  return not contains_vowels
```

Program Text:

```
def find_max_element(my_list):
  """Finds and returns the maximum value contained in my_list.
Assume my_list contains positive values only."""
  max_value = -1
  for i in __________________:
    if ______________________:
      ____________________________________________________________
  return max_value
```