Final Practice Exam

1) Fill in the output in the blanks for the following console code.

```
>>> my_d = {0: 4, 1: 3, 2: 5}
>>> my_l = [5, 4, 3]
>>> my_t = (3, 5, 4)
>>> print(my_d[1])
____
>>> print(my_l[2])
____
>>> print(my_t[3])
____
```

2) What type is each collection? Which of `my_d`, `my_l`, and `my_t` are mutable?

3) Given the following dictionary, answer the following questions:

```
pets = {'Asha': 'cat',
        'Bigby': 'dog',
        'Suvi': 'dog',
        'Tailor': 'newt',
        'Onasis': 'fish'}
```

a) Which are the keys in this dictionary?

b) What are the values?

c) Could we switch the keys and values?

d) How would you find all the names, given a pet species?

e) How would you find the species given a name?

f) How would you add another pet?

4) Suppose you are writing a quizzing program for your own use, to study for an exam in another class. Describe a scenario when you might want to use (there are many possible answers):

a) the dictionary type

b) the list type

c) the tuple

Final Practice Exam

5) Write a function `shift_by(shift_list, offset)` that modifies `shift_list` so that the elements from `offset` to `end` are shifted to the beginning of the list, and the elements from `0 to offset` are move to the end, for example:

```
>>> tmp = ['a', 'b', 'c', 'd']
>>> shift_by(tmp, 2)
>>> tmp
['c', 'd', 'a', 'b']
```

6) Write another function `safe_shift(shift_list, offset)` that returns a new shifted list rather than changing `shift_list directly`.

7) Why would you want two different functions? When would you prefer one over the other?

Final Practice Exam

8) Write a function shoutify(my_text) that returns a modified version of `my_text`, where every capitalized word (except Mr, Ms, and Dr) is replaced with the same word in all capitals.

```
>>> shoutify("Hello there, Dr Evil.")
HELLO there, Dr EVIL.
```

9) Write a wrapper function `shoutify_file(filename)` that applies `shoutify` to the file contents. The function should save the output in a file named the same as the original in all capitals and return `True` if the file was written successfully and `False` otherwise. The function should handle exceptions silently. For example:

```
>>> shoutify_file("/tmp/my_file")
True
>>> shoutify_file("@#$%SE")
False
```

10) Create a custom Exception `FileNameError` for `shoutify_file` to use if the filename passed is already all capitals (thus writing to FILENAME would overwrite the original).

11) What condition would you add to `shoutify_file` to use your new exception appropriately?

Final Practice Exam

12) Consider the following code:

```
x = 0
y = 0
z = 0
M = 4
my_string = "abracadabra"
for c in my_string:
     if c in ['a', 'b']:
          print(c)
          y = y + 1
     elif c=='d':
          print("--")
          continue
     elif y > M:
          break
     else:
          while x < M:
               z = x + z
               x += 1
          print("z"*z)
          x = 0
     print(c.upper())
```

a) How many times is the `c=='d'` condition evaluated?

b) How many times is the `x < M` condition evaluated?

c) How many times is the `print(c)` statement executed?

d) What is the output?

Final Practice Exam

13) Draw the stack trace for the following code:

```
def ponder(y, z):
     print("... ", end="")
     if z >= len(y):
          print("")
          return 0
     else:
          return y[z] + ponder(y, z+1)

def ruminate(z):
     print("Ruminating...")
     if z[0] > z[-1]:
          z[0] = z[0] ** 2
     else:
          z[0] = z[-1]
     return sum(z)

def think(l):
     print("Thinking...")
     return ponder(l, 0)

def consider(pro, con):
     print("Pros:",pro, "and cons:", con)
     p = list(pro)
     c = list(con)
     if sum(p) == sum(c):
          p[0] = ruminate(pro)
          c[-1] = think(con)
          consider(p, c)
     elif sum(p) > sum(c):
          print("YES!")
     else:
          print("NO!")

f1 = [1, 2, -1]
f2 = f1
consider(f1, f2)
print(f1)
print(f2)
```

Answer Key (Dec 7, 2012 Final Practice Exam)

1.
```
>>> print(my_d[1])
3
>>> print(my_l[2])
3
>>> print(my_t[3])
…IndexError…
```

2. my_d -> dictionary, mutable (although the keys must be immutable)
my_l -> list, mutable
my_t -> tuple, immutable

3. a) The keys are 'Asha', 'Bigby', 'Suvi', 'Tailor', 'Onasis', i.e. the pet names.
b) The values are 'cat', 'dog', 'newt', 'fish', i.e. the species
c) Switching the keys and values wouldn't work because there are duplicates in the values.
d) One option:
```
for k, v in pets.items():
    if v=='species':
        print(k) # or append to a list, or write to a file…
```
e) `pets['given_name']` will evaluates to the species of 'given_name' or IndexError if it's not there. You can use `'given_name' in pets.keys()` to check or use `pets.get('given_name')` which returns `None` if it's not there.
f) Several options:
`pets['new_name'] = 'type'` (will overwrite if the name is already in there)
`pets.setdefault('new_name', 'type')` (will return the type 'new_name' is already in there, without replacing it)
`pets.update({'new_name':'type'})` (will replace the type of 'new_name' if it is already there)

4) Many possible answers. The dictionary type maps arbitrary things together, for example vocabulary words and their definitions. The list type when lets you modify the contents, so you might use a list for the words to test, so they can be removed as they are successfully answered. The tuple type is useful when you want a collection that will not be modified. One use would be as the value in the dictionary when multiple answers are possibly correct.

5) Possible solution
```
def shift_by(shift_list, offset):
    shift_list = shift_list[offset:]+shift_list[:offset]
```

6)
```
def safe_shift(shift_list, offset):
    return shift_list[offset:] + shift_list[:offset]
```

7) The two different functions have different postconditions or side-effects. `shift_by` changes the argument and `safe_shift` does not. If you're modifying a temporary list or if memory use is very important you might use `shift_by`. But `safe_shift` is safer for general purposes because it doesn't change anything in the global name space.

8) Possible solution:
```
def shoutify(my_text):
    safe = ['Ms', 'Mr', 'Dr']
    words = my_text.split(" ")
    #this is not careful, if a newline follows a capitalized word
    for i, w in enumerate(words):
        if w[0].isupper() and w not in safe:
            words[i] = w.upper()
    return " ".join(words)
```

9) Possible solution:
```
    def shoutify_file(filename):
        try:
            f = open(filename, 'r')
            lines = f.readlines()
        except IOError:
            return False
        f.close()
        for i in range(len(lines)):
            lines[i] = shoutify(lines[i])

        try:
            f = open(filename.upper(), 'w')
            for l in lines:
                f.write(l)
        except IOError:
            return False
        f.close()
        return True
```

10) Possible:
```
    class FileNameError(Exception):
        pass
```

11) Check:
```
    if filename.isupper():
        raise FileNameError(filename." is taken")
```

12) a)The `c=='d'` condition is checked for every character that isn't 'a' or 'b', before the program breaks.
So it would check for the letters 'r', 'c', 'd', 'r'. The break happens on the last 'r' -> 4 times.
b) The `while` loop runs for the letters 'r', 'c'. The break happens before the while loop on the last r, so the loop runs 2 times in total.
Each time x = 0, then 1, then 2, then 3, then 4 and the condition evaluates to false--> 5 checks
10 times in total.
c) `print( c)` is evaluated for each 'a' and 'b' before the break (which happens on the second 'r'), so six times in total.
d) output:
```
    a
    A
    b
    B
    zzzzzz
    R
    a
    A
    zzzzzzzzzzzz
    C
    a
    A
    --
    a
    A
    b
    B
```

13)
f1 = [1, 2, -1]
f2 = f1 (pointing to the same memory space)
`consider(f1, f2)` is called
    "Pros: [1, 2, -1] and cons: [1, 2, -1]" is printed
    pro and con now point to original list

```
    p = [1, 2, -1] (new memory space)
    c = [1, 2, -1] (new memory space)
    since sum( p) == sum( c)
        ruminate(pro) is called
            z points to the original list (f1)
            "Ruminating…" is printed
            since 1 > -1
            z[0] = 1 --> therefore f1[0] = 1, it is updated with the same value
f1 = [1, 2, -1]
            return 2
        line( p[0] = ruminate(pro) )
        p[0] = 2, so p is updated (but not pro/f1)
    p = [2, 2, -1]
        think(con) is called
            l points to the original list (f1)
            "Thinking…"is printed
            ponder(l, 0) is called
                y points to the original list (f1)
                z = 0
                "…" is printed without a newline
                since 0 < 3
                ponder(y, 1) is called
                    "…" printed
                    ponder(y, 2) is called
                        "…" printed
                        ponder(y, 3) is called
                            z (3) is now >= len(y), so newline is printed
                            return 0
                        line( return y[z] + ponder(y, z+1) )
                        return y[2] + 0 = -1
                    line( return y[z] + ponder(y, z+1) )
                    return y[1] + -1 = 1
                line( return y[z] + ponder(y, z+1) )
                return y[0] + 1 = 2
            line( return ponder(l, 0) )
            return 2
        line ( c[-1] = think(con) )
        c[-1] = 2, so c is updated (but not con/f1)
    c = [1, 2, 2]
    consider(p, c) is called
        pro = p = [2, 2, -1]
        con = c = [1, 2, 2]
        "Pros: [2, 2, -1] and cons: [1, 2, 2]" is printed
        p = [2, 2, -1] (this is a new p, pointing to new memory)
        c = [1, 2, 2] (new memory)
        sum( p) < sum(c ), so "NO!" is printed
        return
    return
```

"[1, 2, -1]" is printed twice because the only change to the f1/f2 memory space was replacing the 1 with a 1.