1. A function is a procedural abstract (a named body of code to perform some action and return a resulting value). The syntax of a function definition is:

```python
def functionName([parameter [, parameter]*]):
  <functionBody>
```

where a parameter can be either:

- formal parameter name
- a keyword argument of the form:  id = value which assigns the formal parameter id a specified default value.
  Note: keyword arguments can only appear as the last parameters in a parameter list
- a pseudo-argument of the form *args that captures all of the remaining non-keyword arguments in a tuple.
- a pseudo-argument keyword argument of the form **kwargs which captures all of the remaining keyword arguments into a dictionary

For example, the function definition “def foo(x, y, *args, **kwargs):” called with “foo(1, 2, 3, 4, a=5, b=6)” will result with formal parameter x containing 1, y containing 2, args containing (3, 4), and kwargs containing {'a':5, 'b':6}. Python uses pass-by-value parameter passing, which copies the value of the actual parameters to the formal parameters. Since variables associated with built-in collections and objects contain references, actual parameters to these only copy their reference values to corresponding formal parameters.

Predict the output of the following Python code segment.

```python
def foo(a, b = 8, *args):
  a = 5
  sum = 0
  for item in args:
    sum += item
  return sum

myInt = 4
myList = [ 1, 2, 3, 4 ]
total = foo(myInt, myList, 10, 11)
print ‘myInt =’, myInt
print ‘myList =’, myList
print ‘total =’, total
```

2. Terminology:

- **scope** - the area of program where an identifier (variable or function name) is known (accessible).
- **lifetime** - the duration of program execution where a variable exists in memory

The namespace of a program is structured in terms of modules. The following table summarizes the kinds of identifiers possible within a module.

<table>
<thead>
<tr>
<th>Kind of Identifier (datum or function)</th>
<th>Location</th>
<th>Scope</th>
<th>Lifetime</th>
</tr>
</thead>
<tbody>
<tr>
<td>module variable (global variable)</td>
<td>Introduced and receives its value at the top level of the module (i.e., outside of all function definitions)</td>
<td>Below its definition at the top level and inside any function definitions</td>
<td>Exist during the whole program’s execution</td>
</tr>
<tr>
<td>parameter (formal parameter)</td>
<td>Introduced in the definition of a function</td>
<td>The body of the function of which it is a parameter</td>
<td>Exist during the lifetime of a particular function call in a call-frame on the run-time stack</td>
</tr>
<tr>
<td>temporary variable (local variable)</td>
<td>Introduced in a function body</td>
<td>The function body below its introduction</td>
<td>Exist during the lifetime of a particular function call in a call-frame on the run-time stack</td>
</tr>
</tbody>
</table>

When two variables with different scope have the same name, the value used is found by looking outward from the inner-most enclosing scope (e.g., a temporary variable’s value it used over a module variable’s value).
Intro. to Computing

a) Draw “boxes” around the different scopes within the following program like was done for the main function.

```python
# File: countDown2.py
# Description: Demonstrates a simple recursive function that takes a specific integer and counts down to 1 before printing "Blast Off!!!"

count = 100

def main():
    print "Start of main...pre-launch stuff"
    doCountDown()
    print "Back in main... control the rocket in flight"

def doCountDown():
    def countDown(count):
        if count == 0:
            print "Blast Off!!!"
        else:
            print count
            countDown(count - 1)

count = input("Enter count down start: ")
print "\nCount Down:"
countDown(count)
print "Done counting down from", count

main()
print "count =", count
countDown(8)
```

b) In each of the above scope boxes including the module level, indicate which identifiers (variable and function) are known/accessable.

c) Explain the run-time error produced when running the above program.

```python
>>> Start of main...pre-launch stuff
Enter count down start: 5

Count Down:
5
4
3
2
1
Blast Off!!!
Done counting down from 5
Back in main... control the rocket in flight
count = 100

Traceback (most recent call last):
  File "C:/Users/fienup/Desktop/Data_Courses/cs051f10/lectures_f10/lec16/countDown2.py", line 29, in <module>
    countDown(8)
NameError: name 'countDown' is not defined
```
3. In Python, functions are *first-class data objects* which means that they can be:
   - assigned to variables,
   - passed as arguments to other functions,
   - returned as the value of another function, or
   - stored in a data structure such as a list or dictionary.

Thus, we write *higher-order functions* that expect a function and a set of data values as arguments. Python has the following predefined higher-order functions with are often useful:

<table>
<thead>
<tr>
<th>Function</th>
<th>General Syntax</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>apply</td>
<td>apply(object[, args[, kwargs]])</td>
<td>apply(pow, (2, 3)) 8</td>
<td>Call a callable object with positional arguments taken from the tuple args, and keyword arguments taken from the optional dictionary kwargs</td>
</tr>
<tr>
<td>map</td>
<td>map(fn, sequence[, sequence, ...])</td>
<td>map(len, ['cat', 'i', 'at']) [3, 1, 2]</td>
<td>Return a list of the results of applying the function to the items of the argument sequence(s).</td>
</tr>
</tbody>
</table>
| filter   | filter(fn or None, sequence) | def odd(n):
| | | return n % 2 == 1
| | | filter(odd, range(9)) [1, 3, 5, 7] | Return those items of sequence for which function(item) is true. If function is None, return the items that are true |
| reduce   | reduce(fn, sequence[, initial]) | def add(x, y):
| | | return x + y
| | | reduce(add, [2, 3, 4, 5]) 14 | Apply a function of two arguments cumulatively to the items of a sequence, from left to right, so as to reduce the sequence to a single value. |

A *lambda* is an anonymous function with no name that can be used to avoid defining a function and then passing it as a parameter to the higher-order functions. The general syntax of a lambda is:

```
lambda <argname-1, argname-2, ..., argname-n> : <expression using argname’s>
```

The reduce example above would be: `reduce(lambda x, y: x + y, [2, 3, 4, 5])`

a) The built-in absolute value function (e.g., `abs(-5)`) takes a number as an argument and return the absolute value of the argument. Write code for a mapping that generates a list of the absolute values of the numbers in a list named `numbers`.

b) Write the code for a filtering that generates a list of the positive numbers in a list named `numbers`. You should first define You should use a lambda to create the auxiliary function.

c) Write the code for a reducing that creates a single string from a list of strings named `words`.