Name:_____

Functions: (Sections 6.1 and 6.2)
A *function* is a procedural abstract, i.e., a named body of code that performs some task when it is called/invoked. Often a function will have one or more parameter that allows it to perform a more general (variable) task. For example, the cube function below can be called with any numeric value with the corresponding cube of that number being returned. NOTE: The def(inition) of the cube function is skipped until called in the print statements.

```
def cube(num):
    '''Function to calculate the cube of a number'''
    num_squared = num * num
    return num_squared * num

# call the function
value = 2
print 'The value', value, 'raised to the power 3 is', cube(value)
print 'The value 3 raised to the power 3 is', cube(3)
```

Terminology:

- a *formal parameter* is the name of the variable used in the function definition. It receives a value when the function is called. In the function cube, `num` is the formal parameter. Formal parameters are only known inside of the function definition. The section of a program where a variable is known is called its *scope*, so the scope of a formal parameter (and other *local variable* defined in the function such as num_squared) is limited to the function in which it is defined.
- an *actual parameter/argument* is the value used in the function call that is sent to the function. In the call to function cube, the variable `value` supplies the actual parameter value of 2.
- a *global variable* is created outside all functions and is known throughout the whole program file, e.g. `value`.

It is helpful to understand the "rules of the game" when a function is called. Memory is used to store the current program and the data associated with it. The memory used to store the data is divided as shown below.

- Global memory is used to store the global variables (and constants).
- The *heap* is used to store dynamically allocated objects as the program runs, e.g. lists and objects
- The *run-time stack* is used to store *call-frames* (or *activation records*) that get *pushed* on the stack when a function is called, and *popped* off the stack when a function returns.

When a function is called the section of code doing the calling is temporarily suspended, and a new call-frames gets pushed on top of the stack before execution of the function body. The call-frame contains the following information about the function being called:

- the *return address* -- the spot in code where the call to the function occurred. This is needed so execution (control) can return there when the end of the function is reached or a `return` statement executes.
- room to store the formal parameters used by the function. In Python, parameters are *passed-by-value* which means that the value of each actual parameter in the function call is assigned to the corresponding formal parameter in the function definition before the function starts executing. However, the memory location for actual parameters for strings, lists, dictionaries, tuples, objects, etc. contain only *references* to the heap
- room to store the local variables defined in the function.

When a function returns, execution resumes at the function call (which is specified by the return address). A function typically sends back a value to the call by specifying an expression after `return` in the return statement. In Python if no expression is specified returned, then the special object `None` is returned.

Name:_____

1. Complete the trace of the below program.

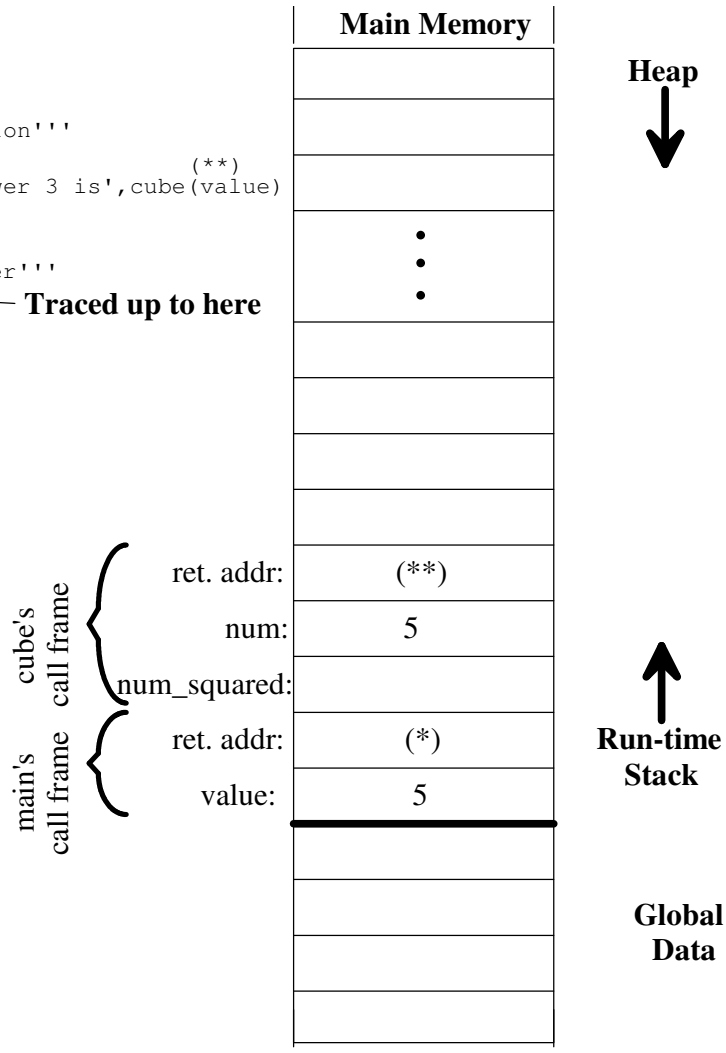
```
''' File:  cube_with_main.py
    Description:  Example of functions
'''
def main():
    '''main function which calls the cube function'''
    value = input("Enter a value to cube: ")
    print 'The value', value, 'raised to the power 3 is',cube(value)

def cube(num):
    '''Function to calculate the cube of a number'''
    num_squared = num * num
    return num_squared * num

# call the main function
main()
#  (*)
```

(**)

Traced up to here



Output of complete program:

```
Enter a value to cube: 5
The value 5 raised to the power 3 is 125
```

2. Write a function that takes as a parameter the Celsius temperature and returns the corresponding Fahrenheit temperature. Recall the formula for the conversion is $F = \frac{9}{5}C + 32$. Call the function using a simple `main` function.

Name:_____

3. Lets work together to perform a top-down design for the following program (Project 11 from Chapter 3.)

In the game of Lucky Sevens, the player rolls a pair of dice. If the dice add up to 7, the player wins $4; otherwise, the player loses $1. Suppose that, to entice the gullible, a casino tells players that there are lots of ways to win: (1, 6), (2, 5), etc. A little mathematical analysis reveals that there are not enough ways to win to make the game worthwhile; however, because many people's eyes glaze over at the first mention of mathematics, your challenge is to write a program that demonstrates the futility of playing the game. Your program should take as input the amount of money that the player wants to put into the pot, and play the game until the pot is empty. At that point, the program should print the number of rolls it took to break the player, as well as maximum amount of money in the pot.

Read the specifications carefully. Try to identify:
a) What would the user's interaction with the program look like?

b) We want the `main` function to act as an outline of the program and contain at most:
- the "main loop": What would be the main loop for this program?

- function calls to perform difficult subproblems. What high-level subproblems does our program need to perform? (Think about what arguments each subprogram needs to be passed or user input needed, and what type of information is returned to the caller)