1. Trace the following program and predict the output.

```
# Program to demonstrate function calls

def main():
    value = 99  # local variable
    print 'In main before call: value is', value
    change_me(value)
    print 'Back in main: value is', value

def change_me(arg):
    value = 10  # local variable
    print 'In change_me: I am changing the formal param.'
    arg = 0
    print 'In change_me: arg is', arg, 'and value is', value
    value = more_change(value, arg)
    print 'Back in change_me: arg is', arg, 'and value is', value

def more_change(arg, arg2):
    print 'In more_change: value is', value
    print 'In more_change: arg is', arg, 'and arg2 is', arg2
    arg = arg2
    print 'In more_change: arg is', arg, 'and arg2 is', arg2
    return arg2 * 5

# Global variable
value = 5
# Call the main function.
main()
```

<table>
<thead>
<tr>
<th>Predicted Output</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Actual Output</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>
2) 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.
- 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 function returns.
- 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.
- 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. Below, trace the same program by building the run-time stack.

```python
# Program to demonstrate function calls
def main():
    value = 99  # local variable
    print 'In main before call: value is', value
    change_me(value)
    print 'Back in main: value is', value

def change_me(arg):
    value = 10  # local variable
    print 'In change_me: I am changing the formal param.'
    arg = 0
    print 'In change_me: arg is', arg, 'and value is', value
    value = more_change(value, arg)
    print 'Back in change_me: arg is', arg, 'and value is', value

def more_change(arg, arg2):
    print 'In more_change: value is', value
    print 'in more_change: arg is', arg, 'and arg2 is', arg2
    arg = arg2
    print 'in more_change: arg is', arg, 'and arg2 is', arg2
    return arg2 * 5

#Global variable
value = 5
# Call the main function.
main()
```

trace the same program by building the run-time stack.