1. Establish three columns:
    1. The Function Call **Stack**, with an initial frame for **Globals**
    2. The **Heap**
    3. Printed **Output**

2. Start at the very top of the code listing and begin by evaluating line-by-line down. When you evaluate a concept on the left, take the action it is associated with it on the right.

Construct	**Action**
Docstring or # Comment	Ignore! Docstrings are documentation for humans. Python effectively ignores them.
Function Definition	1. In the stack's *current working frame*, add the function name and value box. 2. Add a function object to the heap, labelled "Fn Lines *S* - *E*", where *S* is the starting line number of the function definition and *E* is the ending line number. Draw a box around this object. 3. Draw a *reference arrow* from the labeled stack value box *binding* it to the heap object. 4. Ignore the function body! Skip past the indented lines of the function definition body.
print Function Call	Fully evaluate the print function call's *argument expression,* then add the resulting value to the **Output** column of your diagram. You do not need to include quotes around output.
Function Call Expression	1. Focus on the function call's arguments, if they are expressions and not literal values, fully evaluate the argument(s) *first,* from left-to-right, until each argument is a *single value*. 2. Look at the *name* of the function being called in the function call expression. Use *name resolution rules (below)* to confirm this name is bound to a function definition. 3. Focus on the function definition and confirm that the function call's argument(s) evaluated in step 1 exactly match the order and number of parameter(s) *type('s)* in the function definition. • Do not match? Erroneous function call expression! For COMP110's purposes, write down "Function Call Error on Line #C", replacing C with the line the bad function call was encountered, and stop evaluating the program from here. 4. Establish a new function call frame on the stack A. Draw a line separating the *current working frame* from the new call frame B. Add the function's name to the top left of the new frame C. Beneath the function's name, on the left side of the frame, write RA: *C,* where C is the line number the function call being evaluated was written. This is the "Return Address" the program will come back to with the value returned by the function's evaluation. D. On the right-hand side of the new frame, add each of the parameter names of the function on its own line, with some space for its value. E. Copy the fully evaluated argument values from step #1 into its corresponding parameter value. 5. You are now ready to jump to the first line of the function definition's body! The *current working frame* is now the frame you have just established. Work through each statement in the function body line-by-line following the same rules.
Return Statement	1. Return statements can only be found in a function definition body. Anywhere else? Error! 2. Fully evaluate and simplify the *expression* following the `return` keyword to a single value. 3. Beneath RA in the *current working frame*, add the label "RV: ", for "Return Value", and beside of it record the evaluated value you worked out in step #2. 4. Look up the RA (return address) line number of the *current working frame*. You are now ready to jump to this line number and *simplify* or *substitute/replace* the function call expression on this line which you just evaluated, with the Return Value. 5. The *current working frame* on the stack is now the lowest frame on the stack without an RV.
Name Resolution (Function or Parameter Identifiers)	1. Look in the *current working frame* for the name being referenced in an expression. If the name is found here, substitute the value it is bound to for the name in the expression. 2. Not found? Look in the *globals frame* for the name being referenced. If found, substitute. 3. Not found in globals? NameError!
Arithmetic Expressions	Evaluate arithmetic expressions using orders of operations for operators (PEMDAS). In cases of equal precedence, such as Multiplication/Division and Addition/Subtraction, evaluate the expression from left-to-right. Evaluate left-hand operands before right-hand operands.