Computer Science — Worksheet—Code Tracing

BACKGROUND

Computing programs consist of a series of instructions that are executed by a computer. These instructions take data stored in memory and manipulate it to produce the results of the program. Programming requires an understanding of how the computer manipulates data, but there is often a difference in what the programmer *expects* the program she has written will do and what the program instructions *actually* do. The process of correcting a program to produce the desired results is called *debugging*.

One way of debugging consists of performing a *code trace*, following the instructions that one has written one line at a time, and tracking by hand what is happening in memory. In this activity we'll look at a simple program and see how a code trace is performed. Then you'll perform your own code trace on another program provided here.

THE `adder.py` PROGRAM

The source code for the `adder.py` program is given here:

```python
def add(n1, n2):
    """Adds two values and returns result.
    """
    result = n1 + n2
    return result

def main():
    x = 3
    y = 4
    solution = add(x,y)
    print("The sum of",x,"and",y,"is:", solution)

main()

```

If you're very familiar with Python this program will look fairly straightforward: the main program takes the numbers stored in variables `x` and `y` and sends them in as parameter values to the function called `add`. That function adds the numbers together and returns the result to the main program which then prints out the result, in this case the value `7`.

What happens in the computer's memory as this program is running can be shown by following the program's execution line by line and creating a *code trace* diagram off to one side. This code trace identifies which elements of the program are actively being stored in memory and what the values of those elements are.

Let's see how we can create a code trace by hand.

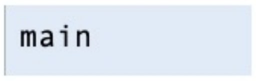
A CODE TRACE EXAMPLE

To perform a manual (paper-and-pencil) code trace I'm going to look at a printout of the program and examine it line by line, even going so far as to place a paperclip or some other physical token next to the line that I'm currently executing—this helps to keep my place and makes sure that I don't skip any lines. As each line gets executed I'm going to write down to the right of the program the state of any memory locations.

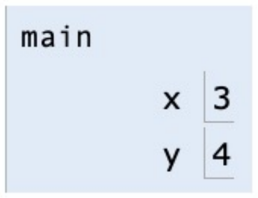
```
 1   def add(n1, n2):
 2       """Adds two values and returns result.
 3       """
 4       result = n1 + n2
 5       return result
 6
 7   def main():
 8       x = 3
 9       y = 4
10       solution = add(x,y)
11       print("The sum of",x,"and",y,"is:", solution)
13
13   main()
14
```

Let's begin.

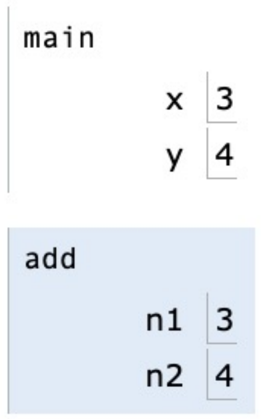
1. I'll begin at line 1, but see that it's a definition for a function `add()` that hasn't been called yet, so I'm going to skip over that section.
2. The next line I could execute is at line 7, but that, too, is a function, `main()`, that hasn't been called yet, so I'm going to skip over that section as well.
3. Ah, line 13 is the first line that I can execute, and it's a call on the `main()` function. I'm going to create a box on the right side of my paper that will track what happens in this `main()` function of the program. I'm also going to move my paperclip/marker up to line 7 of the program to indicate that I have now entered the main function.



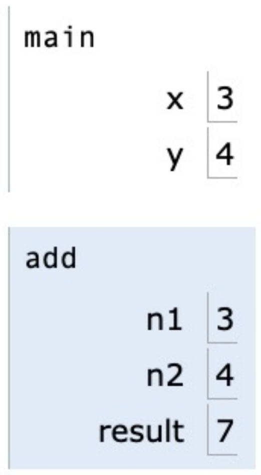
4. Going down to line 8, I see that the variable `x` has been set to the value `3`, so I'm going to record that information in my *main* box. There are a number of strategies for indicating this relationship, but I'm going to use a relatively simple one that simply uses a box with the label `x` to the left and the value `3` that is referred to by that label in the box.
5. Moving my marker down to line 9, I see that `y` is associated with the value `4`. I'm modifying my main box appropriately.



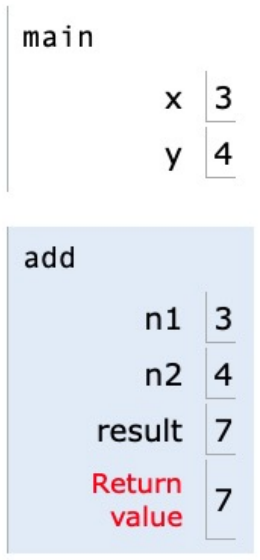
6. Line 10 calls the `add()` function with the values of `x` and y as parameters. In other words, I'm sending in the values of `3` and `4`. Note that when I call this function, those values are assigned to the parameter variables that are listed in the function header at line 1. My main function still has the variables `x` and `y` with the values `3` and `4`, but I also have the `add` function now, with its own variables and their values. Here is what the memory looks like now.



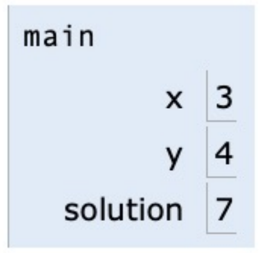
7. We'll skip lines 2 and 3 in the program because they're comments. Line 4 adds the numbers in `n1` and `n2` together and stores them in a new variable called `result`. What does the memory look like now?



8. Moving to line 5, we see that we're going to return the value stored in `result`. We can formally identify that return value in our diagram like this (next page):



9. That return value is sent back to the `main` program, line 10, where the function was called from, and stored in the variable `solution`. The state of the `add` function—its variables and their values—is removed from memory, so our diagram of the computer's memory now just looks like this:



And that's it. For a small-scale problem like this it's almost more trouble than it's worth to go through this process, but if you're having difficulties with a program, being able to do a code trace—either by hand or using a debugger—is a vital skill to have.

ANOTHER CODE TRACE

Here's another program. Perform a pencil-and-paper code trace on this program to identify the states of the program, its variables, and their values as the program runs.

```
"""
quadratic.py
Calculates solutions to quadratic equations
"""

import math

def discriminant(a, b, c):
    """Calculates the discriminant
    """
    d = b * b - 4 * a * c
    return d

def main():
    coeff1 = 1
    coeff2 = -4
    coeff3 = 4
    disc = discriminant(coeff1, coeff2, coeff3)
    if disc == -1:
        print("No real results")
    elif disc == 0:
        print("Single solution: ", -b / (2 * a))
    else:
        print("Two solutions: ", (-b + math.sqrt(disc)) / (2 * a), \
                                 (-b - math.sqrt(disc)) / (2 * a))

main()

```

ADVANCED CODE TRACING

A more complex challenge awaits when you're trying to analyze more dynamic code: a functional program, for example, or a recursive function.

Perform a code trace on this recursive program. You'll need to draw a separate box for each additional function call, even if the previous function hasn't completed running yet.

```python
"""
recursive_fibonacci.py
Has a function that calculates a Fibonacci value recursively.
"""

def fib(n):
    """Recursively calculates the nth Fibonacci value
    """
    if n == 0:
        return 0                    # base case
    elif n == 1:
        return 1                    # another base case
    else:
        return fib(n - 2) + fib(n - 1)      # recursive call

def main():
    print(fib(3))

main()

```