Function Decomposition

Quest

We have come to the end of our two-part saga on return statements, and now, we are ready to put it all together to build a program! Well... almost ready. There is one last concept that we want to touch on before we are ready to start testing and deploying programs on our own. Good programming style includes ensuring that each of your functions has a distinct purpose. The more a single function does, the less reusable it is and the harder it is to understand and debug. This concept is so important that we just had to show it to you (and maybe get to make some cool art while we do it).

Function decomposition is the process of breaking a program into smaller subproblems that make the code easier to read and understand. These subproblems are usually broken down by creating a separate function which the main function can then call. This is especially useful in cases where we have a program that frequently repeats code. In Python, these subproblem functions are called **helper functions** because they help the main program accomplish its task.

Worked Example - IO Art

It's time to make some art! The program below is designed to make "io art" which is a fun way of saying it prints the letters i and o with varying spaces in between. If we vary the number of spaces in the right way, we can make "diamonds." Run this code to see it in action!

```
def main():
    MAX_SPACES = 20
    DIAMONDS = 3

    for i in range(MAX_SPACES):
        print(" ", end="")
    print("io")
    for i in range(DIAMONDS):
        for j in range(1, MAX_SPACES):
            for k in range(MAX_SPACES-j):
                print(" ", end="")
            print("i", end="")
            for k in range(2 * j):
                print(" ", end="")
            print("o")

        for j in range(MAX_SPACES + 1):
            for k in range(j):
                print(" ", end="")
            print("i", end="")
            for k in range(2 * (MAX_SPACES - j)):
                print(" ", end="")
            print("o")


if __name__ == "__main__":
    main()
```

```
                io
               i  o
              i    o
             i      o
            i        o
           i          o
          i            o
         i              o
        i                o
```

▶ Run >_ Hide

This program is pretty cool, but I mean, come on! I'm getting a headache trying to read all those dense for-loops and print statements. We tried to fit that whole program into one function! This is clearly an opportunity to use function decomposition. Let's look at the decomposed version below.

```
def print_n_spaces(n):
    '''
    print_n_spaces prints a row of n spaces on the same line
    params: n (int): number of spaces to print
    '''
    for i in range(n):
        print(" ", end="")  # end="" makes sure the spaces are on the same line


def print_io_line(gap, max_size):
    '''
    print_io_line prints i and o on a line with a certain number of spaces in be

    params:
        gap (int): 1/2 the number of spaces in between i and o
        max_size (int): the maximum number of spaces in a row
    '''

    # Offset io from front of line
    print_n_spaces(max_size - gap)

    # Seperate i and o
    print("i", end="")
    print_n_spaces(2 * gap)
    print("o")


def print_diamond(max_size):
    '''
    print_diamond prints a diamond of io lines where the
    gap in between the i and the o increases until it
    reaches max_size and then decreases to 0

    params:
        max_size (int): the maximum number of spaces in between i and o at
        any one point
    '''
    # Increase gap to max_size
    for i in range(max_size):
        print_io_line(i, max_size)

    # Decrease gap back to 0
```

```
    for i in range(max_size + 1):
        print_io_line(max_size - i, max_size)


def main():
    # How wide should the diamonds be?
    max_size = 20

    # How many diamonds do you want?
    DIAMONDS = 3

    # Print each diamond
    for i in range(DIAMONDS):
        print_diamond(max_size)


if __name__ == "__main__":
    main()
```

▶ Run >_ Hide

Look at how much easier this is to read! The `main` function alone is now only 4 lines of actual code, and it is much easier to understand what it does.

```
def main():
    # How wide should the diamonds be?
    max_size = 20

    # How many diamonds do you want?
    DIAMONDS = 3

    # Print each diamond
    for i in range(DIAMODNS):
        print_diamond(max_size)

if __name__ == "__main__":
    main()
```

You can check out the function header comments to see what each function does. Try to look through each helper function and see how the breakdown of functions works. Which functions are calling which other functions? What is the order of each function call? Understanding this is the key to what makes decomposition so useful.

Catching a Bug

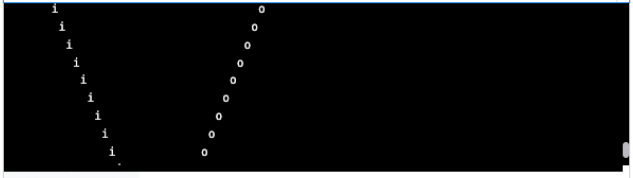
Say, for example, we had a bug in our code where we forgot to add a plus one (also called an **off-by-one error** ⚠️). In the undecomposed code, this would look like this:

```
def main():
    MAX_SPACES = 20
    DIAMONDS = 3

    for i in range(MAX_SPACES):
        print(" ", end="")
    print("io")
    for i in range(DIAMONDS):
        for j in range(1, MAX_SPACES):
            for k in range(MAX_SPACES - j):
                print(" ", end="")
            print("i", end="")
            for k in range(2 * j):
                print(" ", end="")
            print("o")

        for j in range(MAX_SPACES): # deleted the + 1
            for k in range(j):
                print(" ", end="")
            print("i", end="")
            for k in range(2 * (MAX_SPACES - j)):
                print(" ", end="")
            print("o")


if __name__ == "__main__":
    main()
```



▶ Run >_ Hide

If you run this code, you will see that the ends of most of the diamonds are missing! From the code above, without the comment there to show where the error is, would you have been able to figure out which line of code was causing the error? You probably could have (we have faith in you), but it might have taken a *really* long time. With decomposition, this bug is much easier to figure out. Consider the same bug in the decomposed code below:

```
def print_n_spaces(n):
    '''
    print_n_spaces prints a row of n spaces on the same line
    params: n (int): number of spaces to print
```

```
    '''
    for i in range(n):
        print(" ", end="")


def print_io_line(gap, max_size):
    '''
    print_io_line prints i and o on a line with a certain
    number of spaces in between

    params:
        gap (int): 1/2 the number of spaces in between i and o
        max_size (int): the maximum number of spaces in a row
    '''

    # Offset io from front of line
    print_n_spaces(max_size - gap)

    # Separate i and o
    print("i", end="")
    print_n_spaces(2 * gap)
    print("o")


def print_diamond(max_size):
    '''
    print_diamond prints a diamond of io lines where the gap in between the i
    and the o increases until it reaches max_size and then decreases to 0

    params:
        max_size (int): the maximum number of spaces in between i and o at any
        one point
    '''
    # Increase gap to max_size
    for i in range(max_size):
        print_io_line(i, max_size)

    # Decrease gap back to 0
    for i in range(max_size):    # <- off by one error here
        print_io_line(max_size - i, max_size)


def main():
    # How wide should the diamonds be?
    max_size = 20

    # How many diamonds do you want?
    DIAMONDS = 3

    # Print each diamond
    for i in range(DIAMONDS):
        print_diamond(max_size)


if __name__ == "__main__":
    main()
```

```
                    io
                   i  o
                  i    o
                 i      o
                i        o
               i          o
              i            o
             i              o
            i                o
           i                  o
          i                    o
```

▶ Run >_ Hide

Here, the bug is in the `print_diamond` method. Good programming and **debugging** practices involve testing all of our functions and helper functions separately in addition to testing the program as a whole. This way, we could use our separate tests to determine that the helper functions `print_n_spaces` and `print_io_line` work as intended. From there, if we get an error while testing `print_diamond`, we know the error must have occurred within that function and not some other helper function.

The whole program is 20 lines of actual code, but `print_diamond` is only 5 lines. Through decomposition (and good testing practices), you can narrow the hunt for your bug down to a couple of lines of code. As you write larger and more complex programs, this will become even more necessary.

Benefits of Function Decomposition

While function decomposition is not a *requirement* for functional code, it makes code much easier to read and understand. Plus, you can reuse that function in other contexts without rewriting the same code elsewhere in your program. These benefits can save *hours* of debugging and redoing work you've already done. Trust us. Good style is always good practice.