Functions are used for two primary purposes: *computing values* and *side effects*. In the addseq command in the previous section we provide the command with a sequence of numbers and then the command provides us with the sum of the sequence which is a value that we're interested in. In this case we can see that addseq has computed a value based on a few input values. Many other commands, like pwd for example, return a value without affecting the state of the file on our computer. There are however functions like mv or cpwhich move and copy files on our computer. A side effect occurs whenever a function creates or changes files on our computer. These commands don't print any value if they succeed.

We'll often write functions in order to calculate some value, and it's important to understand how to store the result of a function in a variable so that it can be used later. Let's sourceaddseq.sh and run it one more time:

```
source addseq.sh
addseq 3 0 0 7

## 10
```

If we look back at the code for addseq.sh we can see that we created a variable in the function called sum. When you create variables in functions those variables become globally accessible, meaning that even after the program is finished that variable retains its value in your shell. We can easily verify this by echoing the value of sum:

```
echo $sum

## 10

```

This is an example of one strategy we can use to retrieve values that a function has calculated. Unfortunately this approach is problematic because it changes the values of variables that we might be using in our shell. For example if we were storing some other important value in a variable called sum we would destroy that value by accident by running addseq. In order to avoid this problem it's important that we use the local keyword when assigning variables within a function. The local keyword ensures that variables outside of our function are not overwritten by our function. Let's create a new version of addseq called addseq2 which uses localwhen assigning variables.

```bash
#!/usr/bin/env bash
# File: addseq2.sh

function addseq2 {
  local sum=0

  for element in $@
  do
    let sum=sum+$element
  done

  echo $sum
}
```

Now let’s source both files so we demonstrate how local helps us avoid overwriting variables.

```bash
source addseq.sh
source addseq2.sh
sum=4444
addseq 5 10 15 20
echo $sum

## 50
## 50
```

Our original addseq overwrites the value we assigned to sum. Now let’s try addseq2.

```bash
sum=4444
addseq2 5 10 15 20
echo $sum

## 50
## 4444
```

By using local within our function the value of sum is preserved! In order to correctly capture the value of the result of addseq2 we can use command substitution.

```bash
my_sum=$(addseq2 5 10 15 20)
echo $my_sum

## 50
```

Summary

- Functions start with the function keyword followed by the name of the function and curly brackets ({}).
- Functions are small, reusable pieces of code that behave just like commands.
- You can use variables like $1, $2, and $@ in order to provide arguments to functions, just like a Bash script.
- Use the source command in order to read in a Bash script with function definitions so that you can use your functions in your shell.

- Use the local keyword to prevent your function from creating or modifying global variables.
- Be sure to echo the results of your function (if there are any) so that they can be captured with command substitution.

Mark as completed