Learning Objectives - Returning Values

- **Use the `return` keyword to return a value**
- **Identify the return value of the print statement**
- **Demonstrate the ability to return several different data types**
- **Define a side effects**

Returning Values

The Return Keyword

Every function in Python returns a value. Think of the `len` function. This function returns the length (an int) of either a string or list. So the return value of `len` is of type int. `len` does not print anything to the screen, it just returns a number. From here on out, functions will no longer use the print statement. Instead, functions will return a value — use the `return` keyword in place of `print`.

```
def add_five(num):
    """Add five to the parameter num"""
    return(num + 5)

add_five(10)
```

The program no longer prints anything to the screen. That is because the function only adds 5 to whatever parameter is passed to the function, and back the function returns this value to the program. Explicitly tell Python to print the return value of the function.

```
def add_five(num):
    """Add five to the parameter num"""
    return(num + 5)

new_number = add_five(10)
print(new_number)
```

challenge

What happens if you:

- Remove the last two lines of the program and replace them with this: `print(add_five(10))`?

▼ **What is the return value for functions that use `print`?**

If every function in Python has a return value, what is the return value for functions that use `print`? The keyword `return` is not used, so you cannot see if it returns a string, a float, a list, etc. Functions that use `print` instead of `return` have a special return value called `NoneType`. Enter the code below to see the return type of the print statement as compared to the return value of the `len` function.

```
  def print_hello():
      '''Prints the string Hello'''
      print('Hello')

  print(type(print_hello()))
  print(type(len('Hello')))
```

Returning Values

Functions can return any value in Python — ints, floats, strings, lists, etc.

```
def return_int(num1, num2):
    """Return the floor division of num1 divided by num2"""
    return(num1 // num2)

def return_float(num1, num2):
    """Return num1 divided by num2"""
    return(num1 / num2)

def return_string(string):
    """Return the value of string appended to 'Hello' """
    return("Hello" + string)

print(return_int(10, 3))
print(return_float(10, 3))
print(return_string(" friend"))
```

challenge

Can you write a function that returns a list?

If you want to return a list, it is a good idea to have a list be passed as a parameter. Modify the list in some way, and then return it to the program.

▼ **One possible solution**

The code below takes a list of numbers as a parameter. Each element of the list is multiplied by 5, and the new list is returned.

```
def mult_by_5(my_list):
    '''Takes a list of ints and returns a new
    list where each element is multiplied by 5'''
    new_list = []
    for elem in my_list:
        new_list.append(elem * 5)
    return new_list

print(mult_by_5([1, 2, 3, 4, 5]))
```

Side Effects

Side Effects

Side effects are any changes that a function produces that are external to the function itself. Examples of side effects are printing to the screen, changing a global variable, writing to a file, etc. Sometimes side effects are the desired outcome, but there are many programmers who believe that functions should not have side effects (these are called pure functions). The examples below demonstrate how to incorporate pure functions into programs that have side effects.

Modify a Global Variable

Global variables can be modified by a function when using the `global` keyword, which is a side effect. In the for loop, the new value of `my_num` is calculated by calling the function `add_5()`. You can see the value of `my_num` change each time the loop runs.

```
my_num = 0

def add_5():
    """Add 5 to my_num"""
    global my_num
    my_num += 5

for i in range(10):
    add_5()
    print(my_num)

print('The value of my_num is: {}'.format(my_num))
```

challenge

What happens if you:

- Rewrite the function and loop to avoid side effects:

```
my_num = 0

def add_5(num):
    """Receive a number and return that number plus 5"""
    return(num + 5)

for i in range(10):
    print(add_5(i * 5))

print(f'The value of my_num is: {my_num}')
```

▼ **Why the above code is preferred**

Both sets of code print the same sequence of numbers. However, the code where `add_5()` has a parameter is preferable to the code where `add_5()` uses a global variable. If you were to copy/paste the function that relies on a global variable into another program, it would only work if there was a global variable named `my_num`. The function with the parameter, however, will work in another program. Having the parameter means the function is not dependent upon specific global variables. This reduces the chance for an error.

Printing

The code below prints if a number is odd or even. The first function determines if a number is odd or even. The second function constructs the appropriate string. Neither function has a side effect. The act of printing is left to the main program.

```
def is_even(num):
    """Return True if num is even
    return False if num is odd"""
    return num % 2 == 0

def output(num):
    """Return a string with a number,
    and states if that number is even or odd"""
    if is_even(num):
        return "{} is an even number.".format(num)
    else:
        return "{} is an odd number.".format(num)

print(output(2))
```

challenge

What happens if you:

- Change the print statement to `print(output(3))`?

Are Side Effects Bad?

No, side effects are not bad. In fact, they may be the desired result. However, the more side effects a function produces, the greater the risk of introducing a bug. Think about the functions you are writing. If possible, break up your code into several smaller functions, and only introduce side effects when necessary. This may mean you have to write more code, but if this keeps you from having to spend a lot of time debugging, then it is time well spent.

Formative Assessment 1

Formative Assessment 2