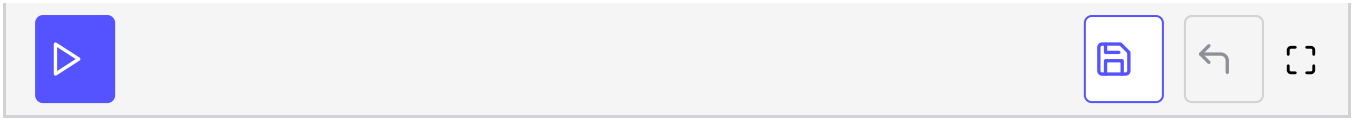
Global Scope

Python includes the **global** statement. It is a keyword of Python. The global statement declares a variable as being available for the code block following the statement. While you can create a name before you declare it global, this is strongly discouraged. Let's attempt to use global to fix our exception from the last example:

```
def my_func(a, b):
    global x
    print(x)
    x = 5
    print(x)

if __name__ == '__main__':
    x = 10
    my_func(1, 2)
    print(x)
```



The output for this code will be this:

```
10
5
5
```

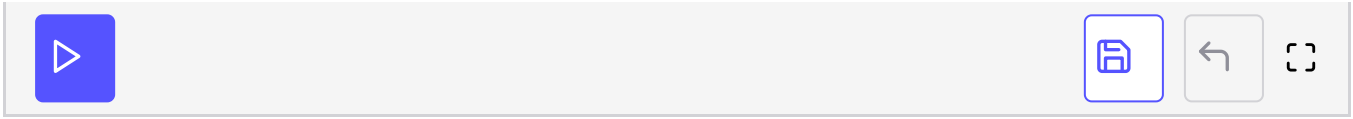
By declaring x to be a global, we tell Python to use the first declaration of x for our first print statement in the function. Then we give x a new value, 5, and print it again before exiting our function. You will notice that since x is now global when we reach the last print statement at the end of the code, x is still 5.

Let's make things extra interesting by mixing globals and locals:

```
def my_func(a, b):
```

```
    global c
    # swap a and b
    b, a = a, b
    d = 'Mike'
    print(a, b, c, d)

a, b, c, d = 1, 2, 'c is global', 4
my_func(1, 2)
print(a, b, c, d)
```



Here we set variable *c* as global. This should make c print out the same both inside and outside our function. We also swap the values of variables a and b in the function to show that we can reassign them inside the function without modifying them outside. This demonstrates that the a and b variables are not global. If you run this code, you should see the following output:

```
2 1 c is global Mike
1 2 c is global 4
```

I just want to note that you shouldn't modify global variables inside of a function. This is considered bad practice by the Python community and it can make debugging quite a bit harder as well.

Now that we understand locals and globals we can move on to learn about **non_local** scope.