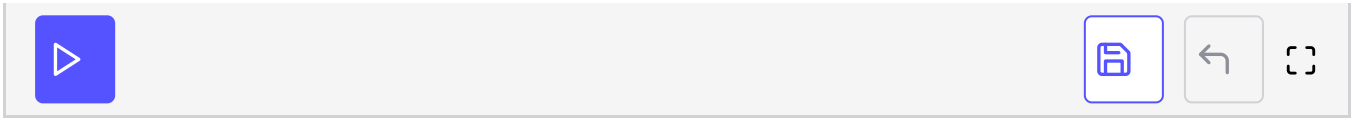
Local Scope

Local scope is the scope you will use the most in Python. When you create a variable in a code block, it will be resolved using the nearest enclosing scope or scopes. The grouping of all these scopes is known as the code blocks *environment*. In other words, all assignments are done in local scope by default. If you want something different, then you'll need to set your variable to **global** or **nonlocal**, which we will be looking at later on in this chapter.

For now, we will create a simple example using Python's interpreter that demonstrates local scope assignment!

```
x = 10
def my_func(a, b):
    print(x)
    print(z)


my_func(1, 2)
#10
#Traceback (most recent call last):
#  File "/usercode/__ed_file.py", line 7, in <module>
#    my_func(1, 2)
#  File "/usercode/__ed_file.py", line 4, in my_func
#    print(z)
#NameError: name 'z' is not defined
```



Here we create variable x and a very simple function that takes two arguments. It then prints x and z. Note that we haven't defined z so when we call the function, we receive a NameError. This happens because z is not defined or is outside the scope. If you define z before you call the function, then z will be found and you won't receive the NameError.

You will also receive a NameError if you try to access a variable that is inside the function only:

```
def my_func(a, b):
    i = 2
    print(x)

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

The variable, i, is only defined inside the function, so when you run this code you will get a NameError.

Let's modify the first example a bit. Put the following into a file and try running it:

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

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

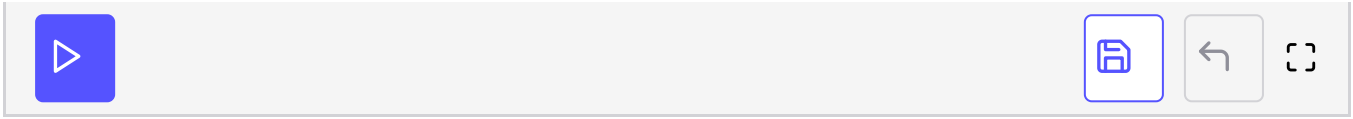
What do you think will happen? Will it print 10 twice? No, it will not. The reason is that we now have two x variables. The x inside of **my_func** has a local function scope and overrides the x variable outside of the function. So when we call my_func, we get 5 printed out instead of 10. Then when the function returns, the x variable inside of my_func is garbage collected and the scope for the outer x comes back into play. This is why the last print statement prints out a 10.

If you want to get really tricky, you can try printing x before we assign it in our function:

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

```
    print(x)

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



When you run this code, you will receive an exception:

```
UnboundLocalError: local variable 'x' referenced before assignment
```

This occurs because Python notices that you are assigning x later on in **my_func** and thus it raises an error because x hasn't been defined yet.