Chapter 8 - User-Defined Functions

User-Defined Functions

Variable Scope

Learning Objectives - Variable Scope

- **Differentiate between global and local scope**
- **Use scope resolution to evaluate functions using global variables**
- **Identify the role of the `global` keyword**

Local Scope

Local Scope

Take a look at the code below. The first function declares the variable `my_var` and then prints it. The second function also prints `my_var`. What do you think the output will be?

```
def function_1():
    my_var = "Hello"
    print(my_var)

def function_2():
    print(my_var)

function_1()
function_2()
```

Python says the problem is that `my_var` is not defined even though the variable is defined on line 3. Variables declared inside a function have local scope. That means `my_var` only “exists” in `function_1`, it cannot be referenced outside of its function. In the image below, light blue box represents the scope of `my_var`. Since `function_2` is outside the scope of `my_var` an error occurs.

```
def function_1():
    my_var = "Hello"
    print(my_var)

def function_2():
    print(my_var)

function_1()
function_2()
```

Local Scope

challenge

What happens if you:

- Change function_2 to look like this:

```
def function_2():
    my_var2 = "Hello"
    print(my_var2)
```

More Local Scope

Each function has its own local scope. That means you can declare two variables with the same name as long as they are in separate

functions. The red my_var exists only in the light red box, and the blue my_var exists only in the light blue box. The boundaries of local scope keep Python from overwriting the value of the first variable with the contents of the second.

Local Scope

```
def function_1():
    my_var = "Hello"
    print(my_var)

def function_2():
    my_var = "Bonjour"
    print(my_var)

function_1()
function_2()
```

challenge

What happens if you:

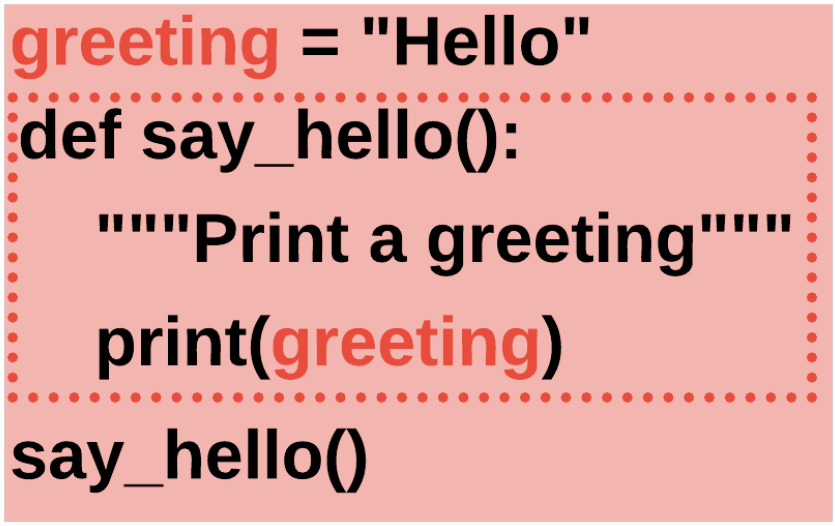
- Declare and call function_3:

```
def function_3():
    my_var = "Hola"
    print(my_var)
```

Global Scope

Global Scope - Referencing Variables

When a variable is declared inside a function, it has local scope. When a variable is declared in the main program, it has global scope. Global variables are declared outside of functions, but can be referenced inside a function.



Global Scope 1

```
greeting = "Hello"

def say_hello():
    """Print a greeting"""
    print(greeting)

say_hello()
```

There is a dotted line around the function because there are limitations on what can be done to global variables.

challenge

What happens if you:

- Modify `greeting` inside the function:

```
greeting = "Hello"

def say_hello():
    """Print a greeting"""
    greeting = "Bonjour"
    print(greeting)

say_hello()
print(greeting)
```

Global Scope - Modifying Variables

The suggestion above asked you to try and modify `greeting` inside the function. However, the output of the program did not change the value of the original `greeting`. Be default, you can reference a global variable in a function, but you cannot modify it. The `global` keyword allows you to modify global variables inside a function. In the image below, there is no more dotted line around the function. `global` removes the restriction for modifying `greeting`. That is why the output is `Bonjour` and `Bonjour`.

```
greeting = "Hello"

def say_hello():
    """Demonstrate how to use
    the global keyword"""
    global greeting
    greeting = "Bonjour"
    print(my_var)

say_hello()
print(greeting)
```

Global Scope 2

```
greeting = "Hello"

def say_hello():
    """Demonstrate how to use the global keyword"""
    global greeting
    greeting = "Bonjour"
    print(greeting)

say_hello()
print(greeting)
```

challenge

What happens if you:

- Make the code look like this:

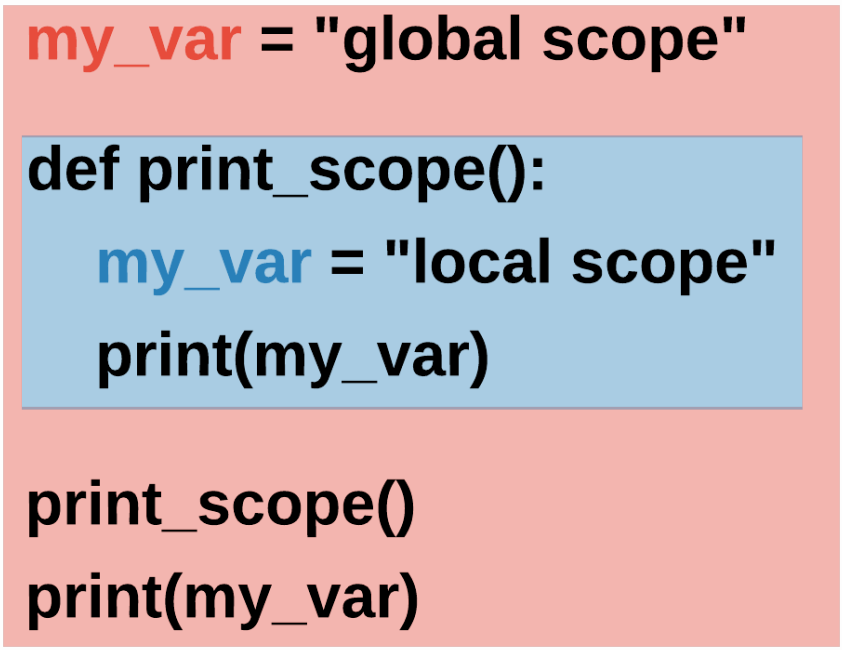
```
def say_hello():
    """Demonstrate how to use the global keyword"""
    global greeting
    greeting = "Bonjour"
    print(greeting)

say_hello()
print(greeting)
```

- Flip the order of `say_hello()` and `print(greeting)`, and run the program again?

Global vs Local Scope

Global vs Local Scope

If there is a collision of local and global variables in a function, the local variable will always take precedence. The global `my_var` (the red one) exists only in the light red area. The local `my_var` (the blue one) exists only in the light blue area. The blue `my_var` is independent of the red `my_var`. That is why the output of the program is two different strings.



Variable Scope

```
my_var = "global scope"

def print_scope():
    """Demonstrate local scope vs global scope"""
    my_var = "local scope"
    print(my_var)

print_scope()
print(my_var)
```

The exception to this rule is when the `global` keyword is being used. In

this case, the global variable takes precedence.

```
my_var = "global scope"

def print_scope():
    """Demonstrate local scope vs global scope"""
    global my_var
    my_var = "local scope"
    print(my_var)

print_scope()
print(my_var)
```

challenge

What happens if you:

- Add the parameter `my_var` to the `print_scope` function and pass `my_var` to `print_scope` in the function call?

```
my_var = "global scope"

def print_scope(my_var):
    """Demonstrate local scope vs global scope"""
    my_var = "local scope"
    print(my_var)

print_scope(my_var)
print(my_var)
```