Scope

Quest

Imagine that with your new powers of function definition, you decide to start writing a super complicated program with lots and lots of functions. You would like to include a variable called `message` at the top of each function that describes what the function is doing. *Wait a second! Don't we need to use a unique name for each variable we create?* If we had to name each of these variables `message_1`, `message_2`, `message_3`, and so on to differentiate them, things would get messy very quickly. Even if we could come up with a clever naming system to easily identify each variable, we'd still have to use a unique word to reference each message. The more functions we add, the more complicated things get. Programmers realized this would be a problem and came up with a solution that is both efficient and adds a bit of security to our programs.

What is Scope

Scope refers to the context in which an element of your code is visible to the rest of the program. If you define a new variable, that variable name can only be used in the same function that it was defined, and that function will be called that variable's scope. Check out this program below:

Error Code

```python
def count_to_n(n):
    '''
    prints numbers from 0 to n-1
    params:
        n (int): number to count up to
    '''
    print(message)
    for i in range(n):
        print(i)


def main():
    message = "Counting up from zero!"
    count_to_n(10)


if __name__ == "__main__":
    main()
```

Run   >_ Show

Try running this code, and you'll see that we get an error. Congratulations, you've just encountered scope! The variable `message` was defined within the `main` function, so it is only visible within `main`. When we try to reference `message` in `count_to_n`, `message` is now **out of scope** and cannot be used.

A variable comes into scope when the program enters into the function where that variable is defined and goes out of scope when you leave that function

Why would we do this? Wouldn't it be better if you could access every variable at any time? Well, recall our example above about having many `message` variables in the same program. We saw that it could actually be more confusing to have all the information at once. The idea of scope is to hide all the variables that aren't relevant to what the program is currently doing. If you define 100 different functions, each with its own descriptive message variable, you probably won't need to know `message_76` if you're in `function_12`, so Python just pretends `message_76` doesn't exist.

When talking about a function, the variables that are in scope within that function are called **local variables**. Which variables are local depends on the function you are talking about, and the local variables change as the program runs through different functions. Below, you can step through a short program and watch variables come in and out of scope:

Click > to see scope in main.py

Reusing Variable Names

Limiting which variables we can access at different times has two remarkably useful benefits. The first is that it allows us to reuse variable names. If we want a descriptive variable at the beginning of each function we define, we don't need a hundred different variable names because they all go out of scope as soon as we leave their respective function. Instead, we can reuse a name like `message` in every function, and that name will be assigned a different meaning depending on which function we are in:

```python
def function1():
    message = "Running function1"
    print(message)

def function2():
    message = "Running function2"
    print(message)

def function3():
    message = "Running function3"
    print(message)

def main():
    function1()
    function2()
    function3()

if __name__ == "__main__":
    main()
```

```
Running function1
Running function2
Running function3
```

▶ Run >_ Hide

Notice how this program prints three different strings to the console, even though we didn't change the name `message` across the three different functions? If scope wasn't a thing, we wouldn't be able to do this.

Function Security

A function is kind of like a restaurant. The customer comes up to the counter and orders some food, the server gives your order to the chef, the chef cooks your food, and the server gives you your meal. Similarly, a function call passes arguments to another function, that function runs some code, and then the function returns to the function call with the result.

In a restaurant, the customer doesn't go into the kitchen and play with the pots and pans, they are just there to order food. Likewise, the function that is calling another function shouldn't be poking around in the other function's body.

If your program contains a function call, the function where that call occurred is referred to as the **caller**, and the function that *gets* called is the **callee**. Consider the following program:

```python
def helloworld():
    '''
    says hello to the world!
    '''
    print("Hello, world!")


def main():
    helloworld()


if __name__ == "__main__":
    main()
```

▶ Run >_ Show

The function `helloworld` gets called within the `main` function. So, in this case, `main` is the caller, and `helloworld` is the callee.

A function call is the only way the caller should interact with the callee. The arguments are the only information the callee has about the caller, and the return value is the only information the caller gets from the callee. This way, we explicitly define how information passes to and from a function, and there are no surprises.

Now we can finally talk about scope. The second reason we love scope so much is that we can leverage it to conceal variables we don't want other parts of the program to use. Any variable defined inside of a function will be out of scope everywhere else, so it is only visible when the program is actively running that function!

Avoid Global Variables

You might be wondering what would happen if you defined a variable outside of a function. What you've created is called a **global variable**. This variable is always in scope because it was defined in the outermost part of the program.

We're actually going to pause here for a moment. Global variables are the first of a few tools you will learn about that you need to be careful about using. Although Python has no problem with you defining global variables, you should be cautious about using them and ask yourself why you have chosen to make one.

We've spent this whole section talking about the beautiful organization that scope provides us. Information is cleanly passed to and from functions and is only useful for a limited amount of time. Global variables completely throw that organization away. You should rarely, if ever, use global variables in your code because using them is horrible style! Global variables can cause inconsistencies within your code.

Look at the example below 🔴

```python
balance = 50.0

def deposit(amount):
    '''
    deposits the amount into the bank account (balance)
    params:
        amount (number): amount of money to deposit
    '''
    balance += amount


def main():
    deposit(10.0)
    print(balance)


if __name__ == '__main__':
    main()
```

▶ Run   >_ Show

Even though `balance` should be in the scope of the function deposit, we get an 🔴 error. This is because when Python gets to the code `balance += amount`, it treats `balance` as a **local variable** to the function `deposit` (a variable defined within the scope of `deposit`) and therefore expects an assignment to have already been made in order to do the += operation.

There are ways to safely use and even modify global variables in your program, but we are just not going to teach you how. We don't want you to get into the habit of defining global variables when you really should have used local variables or passed information as arguments.

Global Constants

The only reason you should ever create a global variable is if you know with full confidence that you will never reassign a new value while the program is running. If you have a constant value that will be useful throughout your entire program, this is the time to use a global variable. We can use global constants to avoid what you might call **magic numbers**, standalone numbers that aren't assigned a name.

Let's look at an example where a global constant might actually be a good idea. In physics, there is an equation for something called gravitational potential energy. It doesn't matter what this actually means; we're just going to write a program that calculates this number using the formula

GPE = mass * gravity * height.

On Earth, **gravity** is approximately equal to **9.8 m/s2** (meter per square second), which we will store as the float 9.8.

Again, we are not focused on the actual physics going on here, just realize that we have this constant value `9.8` that will show up in our program.

So, you write a function called `calculate_gpe` that takes in the mass of an object and its height off the ground and returns the object's gravitational potential energy:

```python
def calculate_gpe(mass, height):
    '''
    function for gravitational potential energy
    params:
        mass (number): mass of object
        height (number): height of object
    return: gpe
    '''
    return mass * 9.8 * height


def main():
    mass = float(input("What is the object's mass? "))
    height = float(input("How high off the ground is the object? "))
    print("Gravitational Potential Energy =", calculate_gpe(mass, height))


if __name__ == "__main__":
    main()
```

▶ Run   >_ Show

If someone else were reading this code, they might see `9.8` and have no idea why it's there or what it represents. Assigning this value to a well-named variable would make the program much easier to read. However, a value like gravity is unique. Many formulas in physics use the value `9.8` to run calculations for objects on Earth, and we know this value is never going to be reassigned. If we wrote functions to calculate some of these other formulas, we'd have to keep defining the same variable over and over again for each function. Finally, we have a good reason to use a global variable.

When naming global constants, we capitalize them to clearly show that they are placeholders for a constant value. We would now define a global constant `GRAVITY` that stores the value `9.8`.

```python
GRAVITY = 9.8

def calculate_gpe(mass, height):
    '''
    function for gravitational potential energy
    params:
        mass (number): mass of object
        height (number): height of object
        return: gpe
    '''
    return mass * GRAVITY * height


def calculate_fgrav(mass):
    '''
    calculates force due to gravity
    params:
        mass (number): mass of object
    return: force of gravity on an object
    '''
    return mass * GRAVITY


def main():
    mass = float(input("What is the object's mass? "))
    height = float(input("How high off the ground is the object? "))
    print("Gravitational Potential Energy =", calculate_gpe(mass, height))
    print("Force due to gravity =", calculate_fgrav(mass))


if __name__ == "__main__":
    main()
```

▶ Run   >_ Show

Now we actually know what that value represents in each function. The other cool thing is that every reference to `GRAVITY` points back to a single variable. We could reassign a new value to `GRAVITY` to the value of gravity on Mars, and suddenly all of our functions would change too. Remember, we aren't allowed to reassign a new value to a global constant *during* a program, but it's fine to adjust the value *before* the program starts.