Table of Contents

Lab 4: Logical Operators, Input Validation, Functions

Learning Outcomes

In this lab, you will:

- learn about input validation,
- use logical operators to test more than on condition on a line,
- learn to create functions,
- be exposed to the concept of scope.

Introduction

What is the goal of programming? A lot of times we create programs in order to automate repetitive tasks. To that end, in this lab we will introduce *functions*, which are used to organize code that will be run several times. Additionally we will look at other methods of writing code that is meant to be easily automated.

Of course, automation is a double-edged sword. Automating tasks means that we can also automate mistakes. In this lab we will also look at ways of testing conditions and ensuring that we are working with good input.

4.1 Input Validation

Let's break the guessing game that we created in the last lab. Run `lab3d.py` and try the following:

```
Guess a number between 1 and 10: gorilla
Traceback (most recent call last):
  File "/home/eric/PRG600/labs/lab3d.py", line 11, in <module>
    ...
ValueError: invalid literal for int() with base 10: 'gorilla'
```

When we try to convert "gorilla" into an integer, Python gets confused and assumes that there has been an error. We have not used *input validation* to test our user's input. In the real world input validation is incredibly important whenever we are working with user-facing input. Not only will users try silly things like 'gorilla', but oftentimes hackers will be testing for vulnerabilities.

Right now what we want to do is avoid all exceptions, if possible. Which means that valid input for this game must adhere to the following rules:

1. It must be numeric. (No gorillas allowed)
2. It must be greater than 0.
3. It must be less than 11.

For the first test, we're in luck: there's a method to check this contained in each string.

```
user_input = input("Enter a number: ")
if user_input.isnumeric():
    print("This only contains characters between 0 and 9.")
else:
    print("This will cause an exception if we try to convert to an integer.")
```

This function is different from `print()` and `round()`, instead of entering something inside the parentheses we attach it to our string with a `.`. This is because `isnumeric()` is a *method* of the string *object*. As you progress with programming, you will learn more about objects. For now, just know that the syntax is slightly different.

For rules 2 and 3, we can simply use comparisons like we have already learned. So three criteria must be met before we try to accept this user input. You might be tempted to use three nested if statements like this:

```
if user_input.isnumeric():
    if int(user_input) > 0:
        if int(user_input) < 11:
            print("This is valid input.")
```

There is a better way.

4.2 Logical Operators

Let's introduce some programming logic: we can link two conditions together by using `and` between them. `and` requires that **both** be true before it will return true. This table is called a *Truth Table* and is widely used to explain boolean logic.

AND

First Condition	Second Condition	Final Result
0	0	0
0	1	0
1	0	0
1	1	1

Notice the only way that 'Final Result' can become '1' is if both conditions equal 1.

Now consider every '0' to be `False`, and every '1' to be `True`.

AND

First Condition	Second Condition	Final Result
False	False	False
False	True	False
True	False	False
True	True	True

This is perfect for our input validation, since we only want to accept the user's guess if it's both a number *and* greater than 0 *and* less than 11.

```
if user_input > 0 and user_input < 11:
    print("Input accepted.")
```

If you *really* want to be fancy, you can use one of Python's improved numerical comparisons as well: `0 < x < 11`. Note that this usually doesn't work in other languages.

```
if user_input.isnumeric() and 0 < int(user_input) < 11:
    print("Input accepted.")
```

here are many other logical operators, but most of what we do can be accomplished with `and`, `or` and `not`.

OR

First Condition	Second Condition	Final Result
0	0	0
0	1	1
1	0	1
1	1	1

Using `or` will return a 'true' if any of the conditions are true.

NOT

First Condition	Final Result
0	1
1	0

`not` will invert a true/false.

4.3 TASK: Guessing Game Improved.

- Copy your `lab3d.py` file and rename it to `lab4a.py`
- Ensure that the guessing game will validate input from the user. Each guess needs to be a number between 1 and 10.
- Print a useful error message for the user if the input is not a number, or if the input is numeric but is out of bounds.

Sample Output

```
Enter a number between 1 and 10: 1
Sorry, try again.
Enter a number between 1 and 10: hamburger
Error: not a number or out of bounds.
Enter a number between 1 and 10: 100
Error: not a number or out of bounds.
Enter a number between 1 and 10: 6
Correct! You win
```

4.4 What Is A Function?

Functions in programming are derived from functions in mathematics. You are probably familiar with the following syntax:

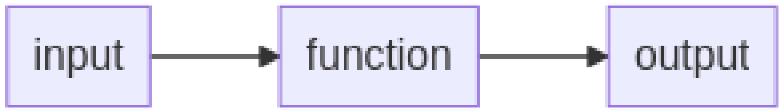
$f(x) = x^2 + 3x + 1$

The idea with this is that we can define **x** differently and get a different value returned in each case.

$f(3) = (3)^2 + 3(3) + 1$

$f(3) = 19$

The input changes, and the output changes. But the relationship between input and output, defined by the equation, does not change.



In programming, we are often performing identical processing on many different inputs. Consider an example where we are processing many files in a similar way.

Another example would be in computer graphics, where we need to perform many complex calculations in order to correctly render light reflecting off a window. Each frame of animation is going to need to run this function, with a different result each time.

This is where we would make use of a function. To implement the example above, we would enter it like this:

```
def calculation(x):
    "an example of a calculation"
    return x ** 2 + (3 * x) + 1
```

- The `def` line is called a *function definition*. Here we can name the function, and name its *parameters*.
- The next line is a *function-level docstring*. This is a comment that defines what the function will do. Just like the the in-line comments that you've already seen, this is not executed but is useful for people reading your code.
- The `return` line defines what data your function will produce. As soon as Python encounters a `return` instruction, it will stop running code in your function!

```
def calculation(x):
    "an example of a calculation"
    return x ** 2 + (3 * x) + 1  # after this line, we exit the function and
return to the main script.
    print("This will never get printed.")
```

Calling a Function

Once you've defined a function, you're only halfway finished. You will have to *call* the function inside your code. Here is an example of this using the Python interpreter. We have already typed in our function definition.

```
>>>> y = calculation(3)
>>>> print(y)
19
>>>> print(calculation(4))
29
```

In the first example, we are storing the output *returned* by the function into a variable called y. The second example combines both calling the function and printing its return value into one step. This sort of combination of steps is common, but can lead to confusion. When things go wrong, be sure to break your instructions into more steps (or use the debugger).

Guidelines For Using Functions

The code that you eventually create will be doing different things, so it's hard to provide 'golden rules' that are always going to be relevant. However, There are some general good practices that you should follow:

- Functions don't have to be mathematical! Whenever you have one or more lines of code that will be repeated, you can put these into a function.
- Functions should be short, and as much as possible should do **one thing well**.
- Function names should contain a verb. Functions *do things*. Functions should use lower-case words and underscores.
- Functions can contain many parameters, or none. Functions can return either one piece of data, or nothing.
- Functions do not return multiple values. Generally, if you are in the situation where you want a function to return two variables, you should consider creating more functions.

4.5 TASK: Function Examples

Look at the following code below. Each function is adhering to general best practices. For each function:

1. provide a *function-level docstring* so that a Python beginner will be able to understand what the function is doing.
2. use an *in-line comment* at the function definition to tell me how many parameters there are, and if there is a return value.
3. call the function using your own *arguments*. Test the code and makes sure you don't have any exceptions.

When you're finished, save your code as `lab4b.py`.

```python
from random import randint

def rtrn_area(length, width):  # Put your in-line comment here
    "Put your function level docstring here"
    return length * width

rect = rtrn_area(5, 3)
# call the function again with new values

def print_all_caps(name, age):  # Put your in-line comment here
    "Put your function level docstring here"
    cap_name = name.upper()
    print('THIS PERSON\'S NAME IS ' + cap_name + ' AND THEY ARE ' + str(age)
+ ' YEARS OLD!!!')

print_all_caps('eric', 41)
print_all_caps('melissa', 40)
# call the function again with new values

def get_rando():  # Put your in-line comment here
    "Put your function level docstring here"
    return randint(1, 101)

lucky_num = get_rando()
# call the function again with new values

def is_odd(num):  # Put your in-line comment here
    "Put your function level docstring here"
    if num % 2 == 1:
        return True
    else:
        return False

print(is_odd(13))
print(is_odd(get_rando()))
# call the function again with new values
```

4.6 Portable Functions

Portability refers to being able to run functions in a new Python program, by importing functions in much the same way we would import functions from the standard library.

In a new file called `importer.py`, enter the following code. Make sure that `importer` and `lab4b` are in the **same directory**.

```
from lab4b import is_odd

print(is_odd(27))
```

What you will notice, however, is all of the print statements and function calls you created in lab4b.py are run when you imported the file. What we need is a way to prevent code from running when the file is being *imported* versus when it is *executed*.

Open `lab4b.py` again. After the functions definitions, put the following line:

```
if __name__ == "__main__":
    print(is_odd(13))
    # put the rest of your function calls here.
```

- Save and run `lab4b.py` again. You will notice that when you run the script as a standalone file, the lines inside `__name__ == "__main__"` are executed. We can call this block *main*.
- Close `lab4b.py` and run `importer.py` again. You should see no output printed to the screen whatsoever. Check that function calls are now in *main*.

The other aspect of making functions portable is making sure they can be run without needed to access any variables that are *global*, that is, declared outside of the function itself. Any variables used inside the function should be passed in as arguments.

4.7 Arguments Versus Parameters

You might be confused about these terms, they both seem to refer to the 'input' of functions. Are they the same?

No. Parameters are part of the *function definition*, and define variables *inside* the function. Parameters are considered *local*. In the example below, `diameter` is considered a parameter. (Also note that we are importing the value of Pi from another useful Python module).

```
import math

def circle_area(diameter):
    return math.pi * diameter ** 2
```

Arguments are part of a *function call*. Their scope is considered *global*. These are values that are outside the function. In the example below, the number `26` is considered an argument.

```
area = circle_area(26)
```

This introduces us to an important concept called *scope*.

What is Scope?

Scope refers to where a variable exists. Functions are 'born' when they are called, and 'die' when they return a value or reach the end of their code block. The 'scope' of their parameters is inside the function, and when the function 'dies' the value of those parameters are lost. Similarly, when you define variables outside of functions, they are considered to have *global* scope. The value of those variables are lost only when the program exits.

One confusing issue is that local and global variables can have the same name, but still be considered completely different variables with different values. To illustrate this, be sure to use VSCode to run the example below. Put a **breakpoint** on the line with `area = 0` and step through the code as we demonstrated in the last lab.

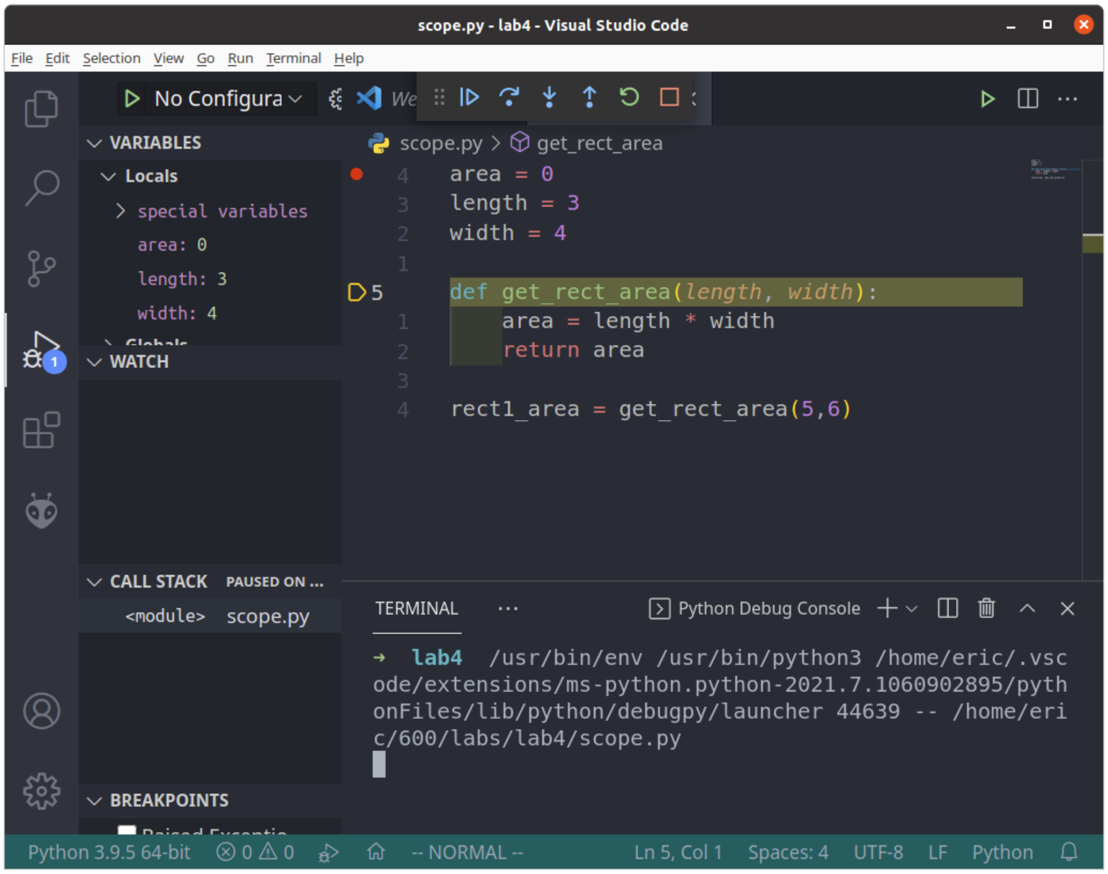
```
area = 0
length = 3
```

```
width = 4

def get_rect_area(length, width)
    area = length * width
    return area

rect1_area = get_rect_area(5,6)
print("Complete")
```

As you step through the code, you will see `area`, `length` and `width` defined in the 'Variables' box.


scope.py - lab4 - Visual Studio Code
File Edit Selection View Go Run Terminal Help
No Configura
VARIABLES
Locals
special variables
WATCH
CALL STACK PAUSED ON ...
<module> scope.py
BREAKPOINTS
TERMINAL
Python Debug Console
Python 3.9.5 64-bit ⊗ 0 ⚠ 0 -- NORMAL -- Ln 5, Col 1 Spaces: 4 UTF-8 LF Python

```
area: 0
length: 3
width: 4
```

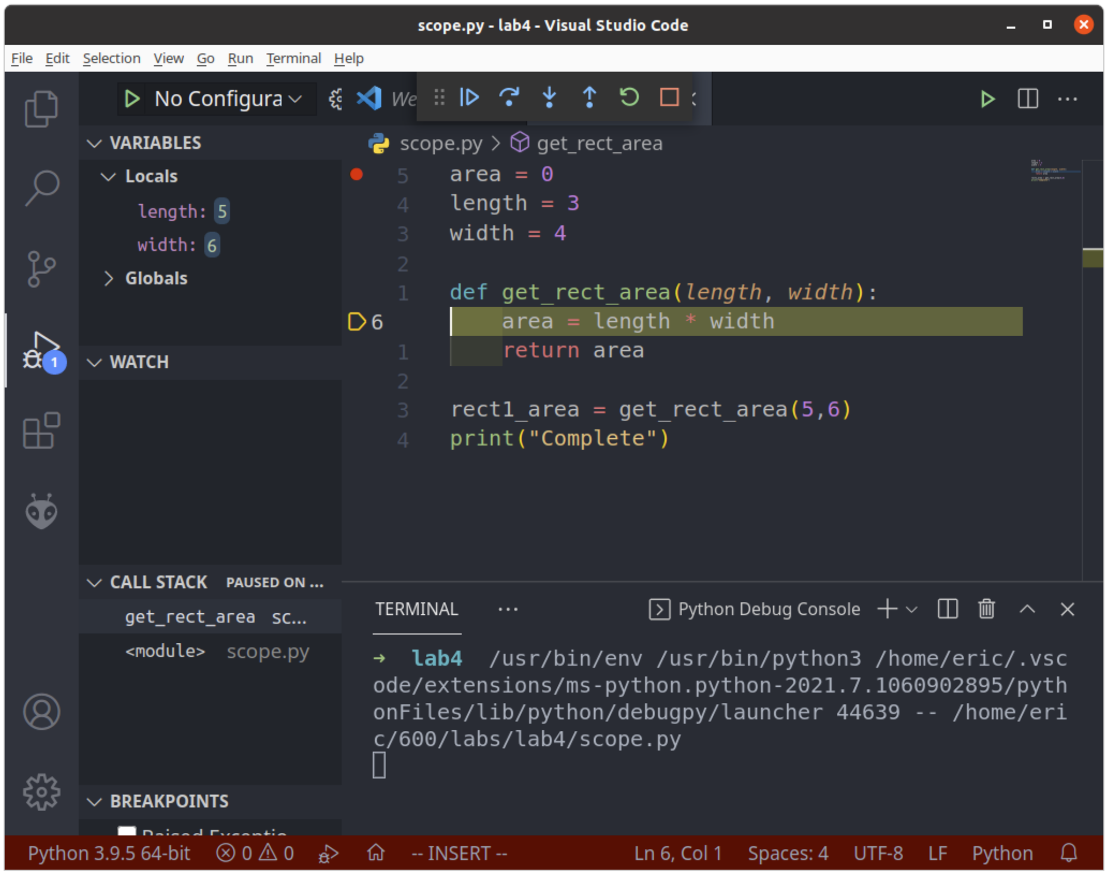
```
area = 0
length = 3
width = 4

def get_rect_area(length, width):
    area = length * width
    return area

rect1_area = get_rect_area(5,6)
```

```
➜  lab4  /usr/bin/env /usr/bin/python3 /home/eric/.vscode/extensions/ms-python.python-2021.7.1060902895/pythonFiles/lib/python/debugpy/launcher 44639 -- /home/eric/600/labs/lab4/scope.py
```

When you reach the function call, make sure to use the **Step Into** button rather than Step Over. "Step Into" will enter into a function, whereas "Step Over" will skip over it.



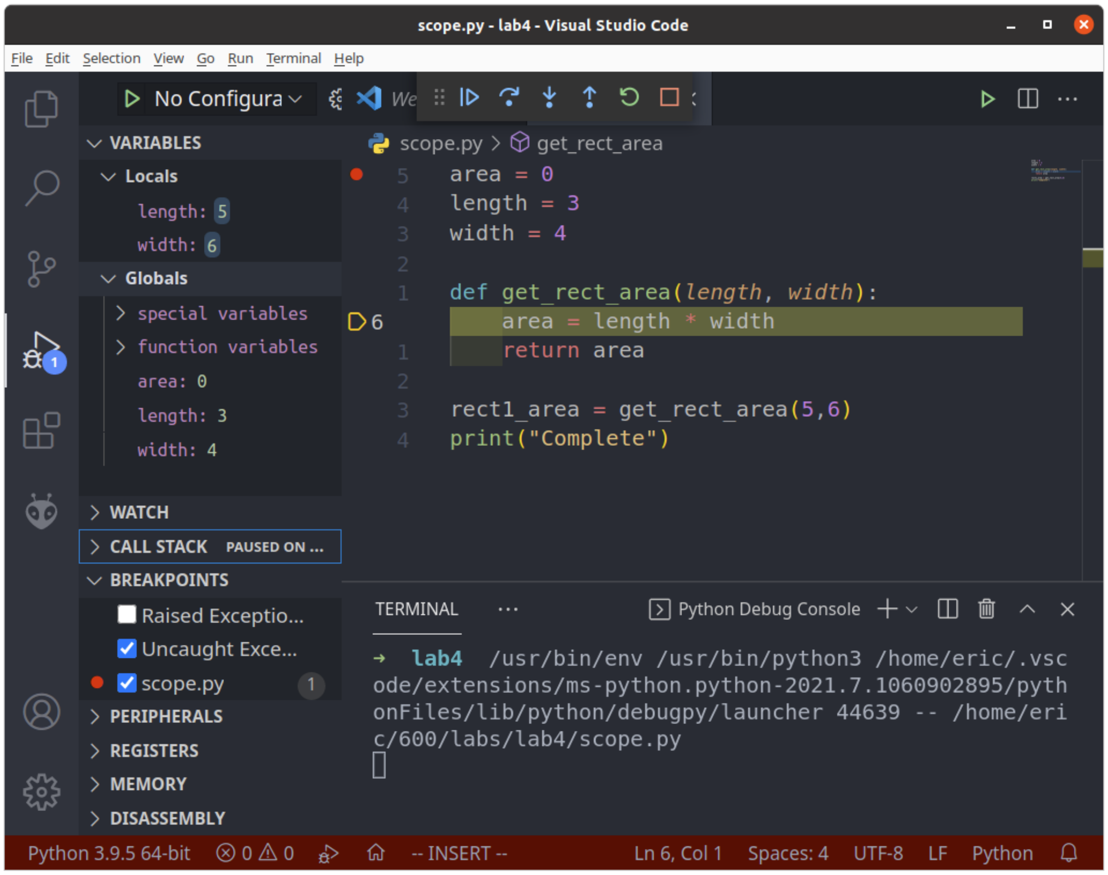
```
area = 0
length = 3
width = 4

def get_rect_area(length, width):
    area = length * width
    return area

rect1_area = get_rect_area(5,6)
print("Complete")
```

```
➜  lab4  /usr/bin/env /usr/bin/python3 /home/eric/.vscode/extensions/ms-python.python-2021.7.1060902895/pythonFiles/lib/python/debugpy/launcher 44639 -- /home/eric/600/labs/lab4/scope.py
```

In the Variables box, you will see that the values for `length` and `width` have changed. What happened to the previous values? The answer is that they are still there, only they aren’t visible at the moment. Click on the “Globals” Drop-Down.



```
Locals
  length: 5
  width: 6
Globals
  > special variables
  > function variables
  area: 0
  length: 3
  width: 4
```

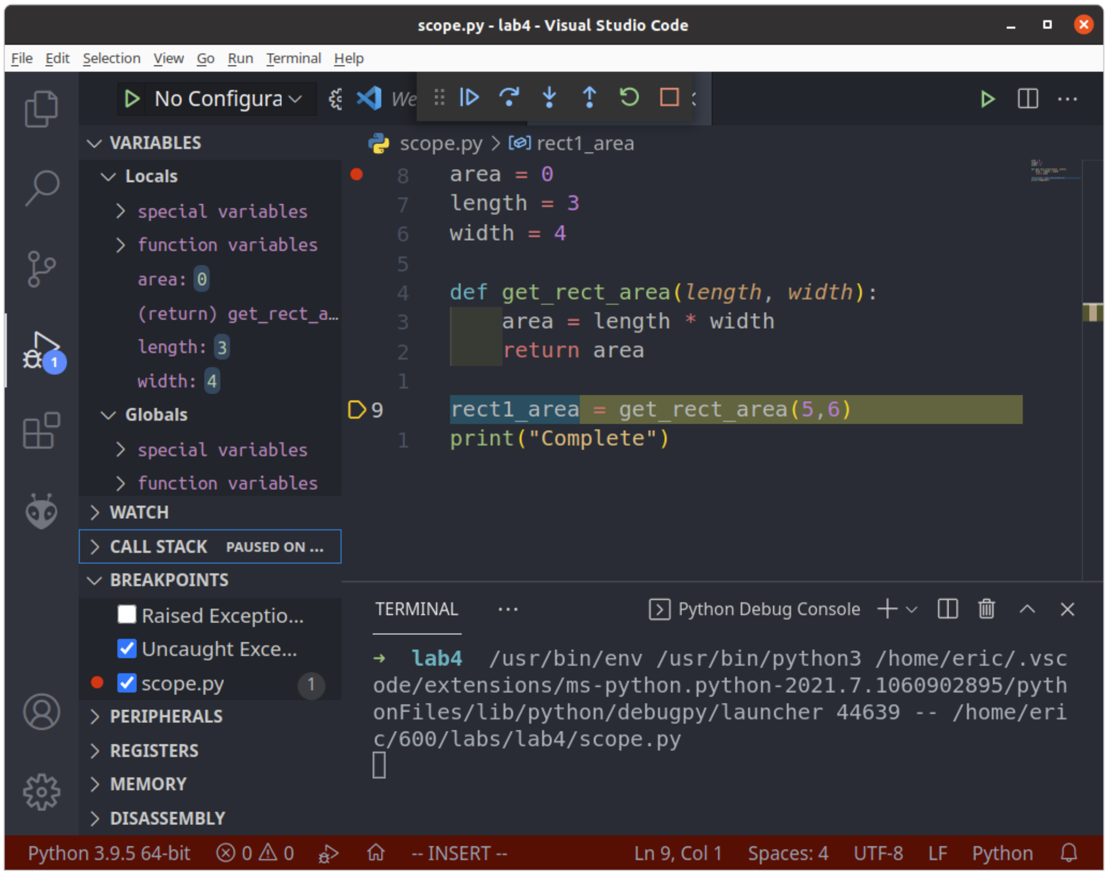
```python
area = 0
length = 3
width = 4

def get_rect_area(length, width):
    area = length * width
    return area

rect1_area = get_rect_area(5,6)
print("Complete")
```

```
→  lab4  /usr/bin/env /usr/bin/python3 /home/eric/.vsc
ode/extensions/ms-python.python-2021.7.1060902895/pyth
onFiles/lib/python/debugpy/launcher 44639 -- /home/eri
c/600/labs/lab4/scope.py
```

There exist **two** `lengths`: one local (equals 5) and one global (equals 3). What happens inside the function, stays inside the function. Go ahead and use “Step Over” or “Step Through” to finish running the code.



```
Locals
  > special variables
  > function variables
    area: 0
    (return) get_rect_a...
    length: 3
    width: 4
Globals
  > special variables
  > function variables
```

```python
area = 0
length = 3
width = 4

def get_rect_area(length, width):
    area = length * width
    return area

rect1_area = get_rect_area(5,6)
print("Complete")
```

```
→  lab4  /usr/bin/env /usr/bin/python3 /home/eric/.vsc
ode/extensions/ms-python.python-2021.7.1060902895/pyth
onFiles/lib/python/debugpy/launcher 44639 -- /home/eri
c/600/labs/lab4/scope.py
```

Once we exit the function, the local version of `length` `width` and `area` revert to their previous values. The function has died, and the paramter values are lost. Only the return value is preserved, and this is assigned to the variable `rect1_area`.

What should we learn from this example?

- Scope is important.
- Debuggers are useful.
- It's always better to give variables *descriptive names*. When you have ten different variables all called x, debugging your code becomes a nightmare.

4.8 TASK: Circle Area Calculator

Let's say that your company, for whatever reason, needs a tool to return many areas of circles using a command line tool. Here are the requirements:

- The program is required to have a function called `circle_area` that has `radius` as its parameter and returns the calculated area.
- The program is required to run the rest of the code inside of a **main block** (`if __name__ == "__main__":`).
- Inside the main block:

- The program should prompt the user to enter numbers. The numbers should be integers, and between 0 and 1999.
- If the user enters an input that is not an integer, or is out of bounds, the program should print an error message and repeat the prompt.
- Each number is considered to be the radius of a circle. For each circle, print both the *radius* and the *area* (Create a function to return circle area).
- The user should be able to continue entering numbers until they press Enter to stop.
- However, if the user presses Enter at the first prompt, print the message: `Program was cancelled.`

Sample Output

```
Circle Area Calculator
Enter a radius between 0 and 1999. Press Enter to exit: 2
Radius: 2. Area: 12.566370614359172.
Enter a radius between 0 and 1999. Press Enter to exit: ten
Error: ten is not a number.
Enter a radius between 0 and 1999. Press Enter to exit: 17
Radius: 17. Area: 907.9202768874502.
Enter a radius between 0 and 1999. Press Enter to exit: 2001
Error: 2001 is out of range.
Enter a radius between 0 and 1999. Press Enter to exit:
Exiting...
```

Save the file as `lab4c.py` and submit it as part of your lab.

CHALLENGE TASK: Calculate Trend

- Save this file as `challenge4.py`.
- Create a function called `rtrn_slope`. This function will take 4 integers and return a float.
- each float will be between 0 and 10. You will do input validation *outside* of the function.
- Inside the function, the first two numbers are the x and y values of a point on a Cartesian map. **Note:** You can review Cartesian maps on Khan Academy, if you don't remember!
- The last two numbers are the x and y values of a second point.
- Your function should return the calculate slope of these two points.
- Call your `rtrn_slope` function inside your `challenge4.py` script. Use *hard-coded* values to test. For example, `rtrn_slope(1, 1, 3, 3)` should return a slope of `1.0`.
- Once your function is returning the correct slope, use `input()` to get these values from the user. Remember to **validate the user input**!. Values outside of the acceptable range should cause an error.
- Finally, put the second `input()` statement inside a while loop. Your script should now calculate the slope of the *previous value* along with the *current value*. You can create the option to quit if you wish, or use Ctrl+d to terminate the program at any time.

Sample Output

Please note I omitted the input validation step from this sample, however your code should have it!

```
Enter the starting X value: 1
Enter the next Y value: 1
Enter the new X value: 3
Enter the new Y value: 3
Slope is: 1.0
Enter the new X value: 3
Enter the new Y value: 3
Slope is: 0.0
Enter the new X value: 5
Enter the new Y value: 4
Slope is: 0.5
Enter the new X value: 0
Enter the new Y value: 1
Slope is: 0.6
Enter the new X value: 3
Enter the new Y value: 0
Slope is: -0.3333333333333333
```

Deliverables

You can download the check script here.

- ☐ `lab4a.py`
- ☐ `lab4b.py`
- ☐ `lab4c.py`
- ☐ `lab4-check-output.txt`
- ☐ `challenge4.py`

Glossary

- Function
- Argument
- Parameter
- Function Definition
- Function Call
- Scope
- Local/Global
- Input Validation
- Boolean Logic
- Truth Table