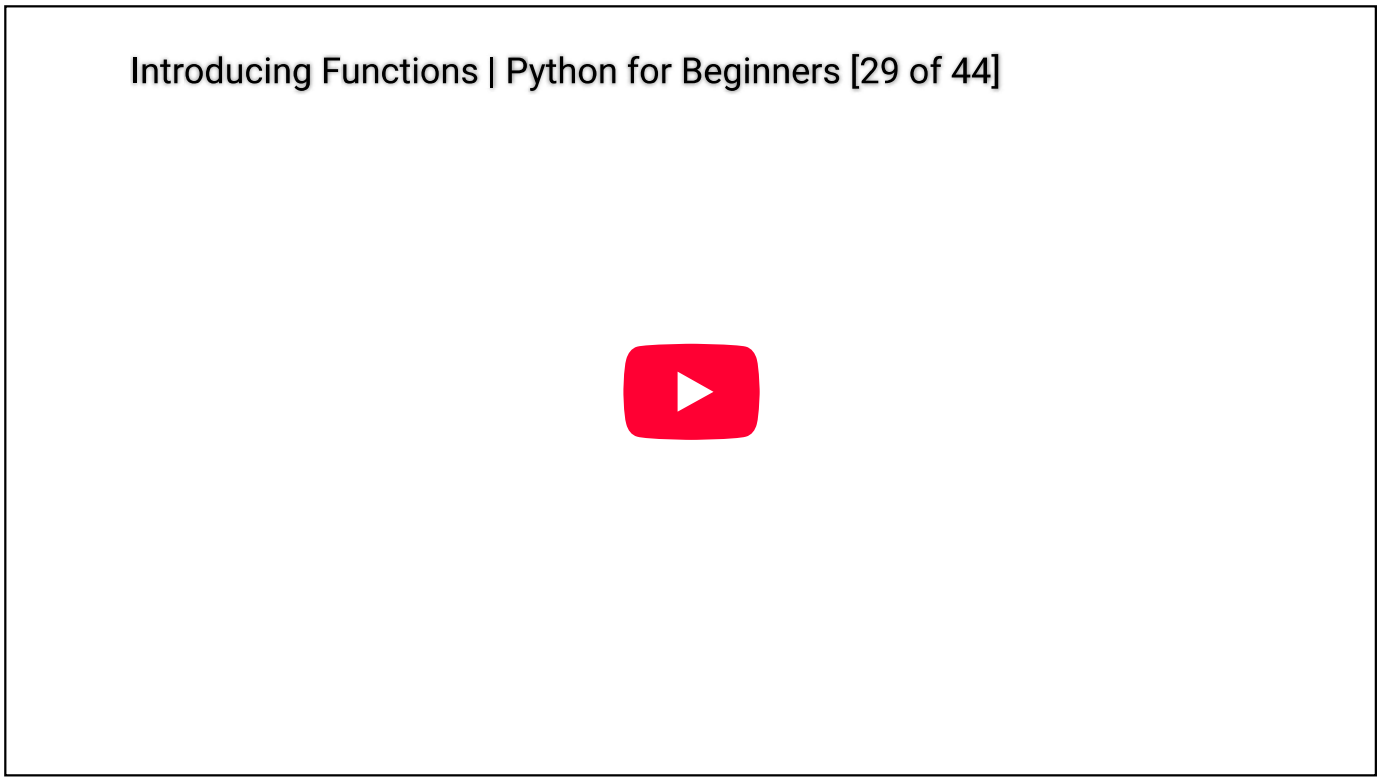
W02 Learning Activity (1 of 2): Writing Functions

Because most useful computer programs are very large, programmers divide their programs into parts. Dividing a program into parts makes it easier to write, debug, and understand the program. A programmer can divide a Python program into modules, classes, and functions. In this lesson and the next, you will learn how to write your own functions.

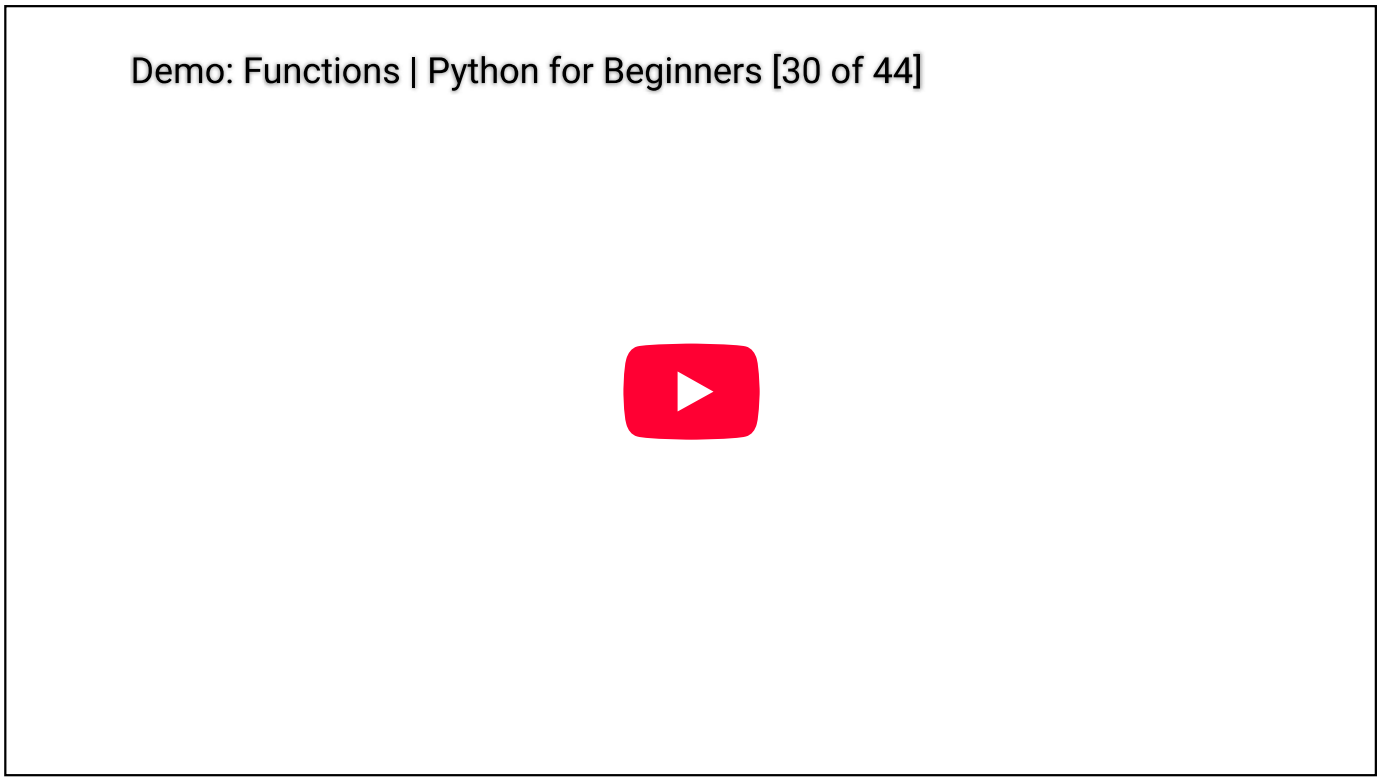
Videos

Watch the following four videos from Microsoft about writing functions:

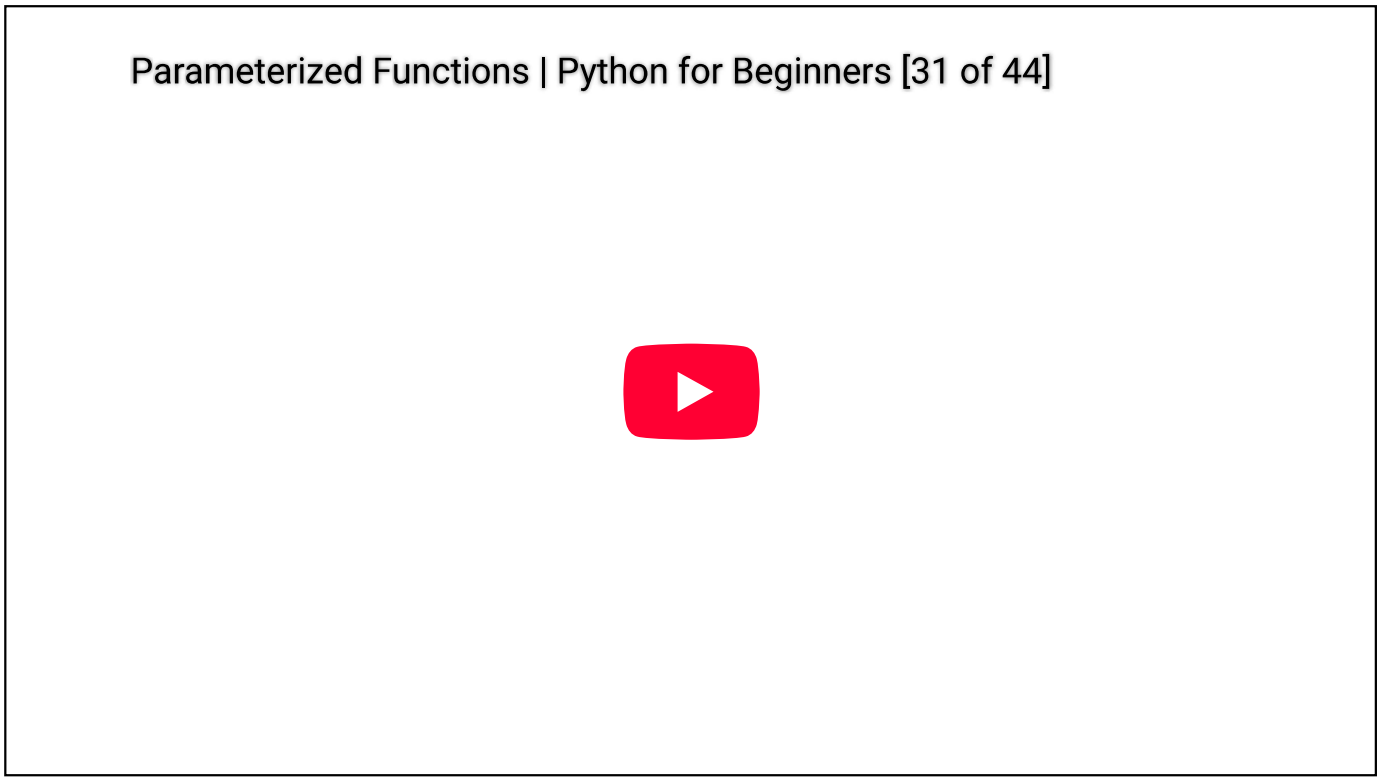
1. Introducing Functions (10 minutes)



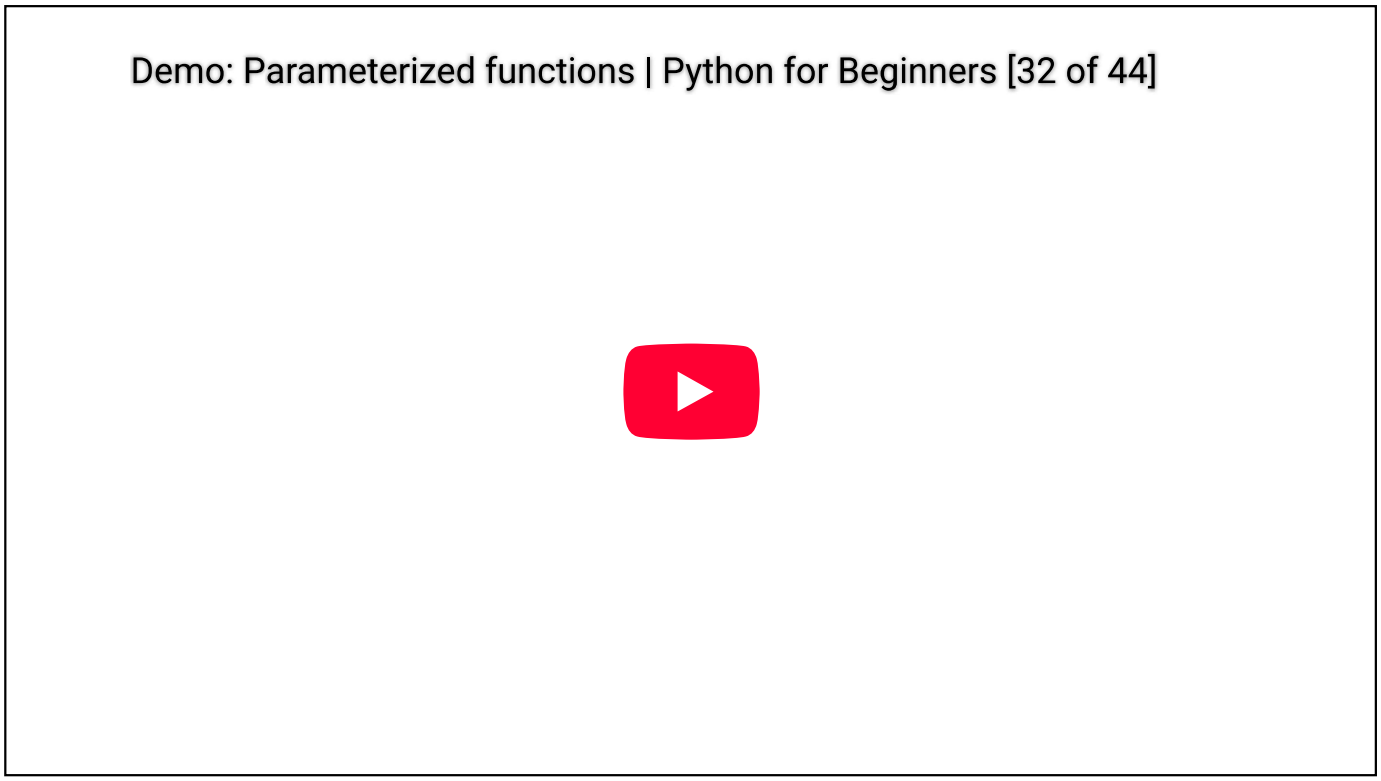
2. Demonstration: Functions (8 minutes)



3. Parameterized Functions (7 minutes)



4. Demonstration: Parameterized Functions (5 minutes)



Concepts

Here are the Python programming concepts and topics that you should learn during this lesson.

What Is a Function?

A *function* is a group of statements that together perform one task. Broadly speaking, there are four types of functions in Python which are:

1. Built-in functions
2. Standard library functions
3. Third-party functions
4. User-defined functions

In the previous lesson, you learned how to call the first two types of functions. In week 3, you will learn how to install third-party modules and call third-party functions. In this lesson, you will learn how to write and call user-defined functions.

What Is a User-Defined Function?

A *user-defined function* is a function that is not a built-in function, a standard function, or a third-party function. A user-defined function is written by a programmer like yourself as part of a program. For some students the term “user-defined function”

is confusing because the user of a program doesn't define the function. Instead, the programmer (you) define user-defined functions. Perhaps a more correct term is programmer-defined function. Writing user-defined functions has several advantages, including:

1. making your code more reusable
2. making your code easier to understand and debug
3. making your code easier to change and add capabilities

How to Write a User-Defined Function

To write a user-defined function in Python, simply type code that matches this template:

```
def function_name(param1, param2, …
  paramN):
  """documentation string"""
  statement1
  statement2
  ⋮
  statementN
  return value
```

The first line of a function is called the *header* or *signature*, and it includes the following:

1. the keyword **`def`** (which is an abbreviation for "define")
2. the function name
3. the parameter list (with the parameters separated by commas)

Here is the header for a function named **`draw_circle`** that takes three parameters named **`x`**, **`y`**, and **`radius`**:

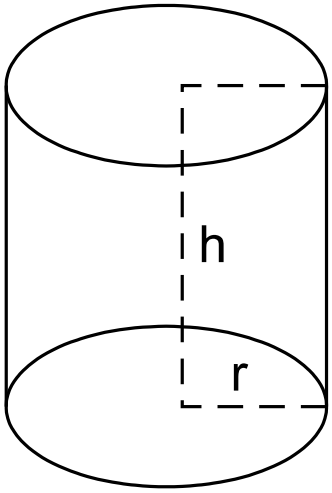
```
def draw_circle(x, y, radius):
```

You could read the previous line of code as, "Define a function named **`draw_circle`** that takes three parameters named *x*, *y*, and *radius*."

The *function name* must start with a letter or the underscore (_). The rest of the name must be made of letters, digits (0–9), or the underscore. A function name cannot

include spaces or other punctuation. A function name should be meaningful and should describe briefly what the function does. Well-named functions often start with a verb.

The statements inside a function are called the *body* of the function. Just like other block statements in Python, such as **`if`**, **`else`**, **`while`**, and **`for`**, all of which end with a colon (:), you must indent the statements inside the body of a function. The body of a function should begin with a *documentation string* which is a triple quoted string that describes the function's purpose, parameters and return value. The body of a function may contain as many statements as you wish to write inside of it. However, it is a good idea to limit functions to less than 20 lines of code.



A right circular cylinder with radius *r* and height *h*

Example 1 contains a function named **`print_cylinder_volume()`** with no parameters that gets two numbers from the user: *radius* and *height* and uses those numbers to compute the volume of a right circular cylinder and then prints the volume for the user to see.

```
# Example 1
import math
# Define a function named print_cylinder_volume.
def print_cylinder_volume():
    """Compute and print the volume of a cylinder.
    Parameters: none
    Return: nothing
    """
    # Get the radius and height from the user.
    radius = float(input("Enter the radius of a cylinder: "))
    height = float(input("Enter the height of a cylinder: "))
    # Compute the volume of the cylinder.
```

```
  volume = math.pi * radius**2 * height
  # Print the volume of the cylinder.
  print(f"Volume: {volume:.2f}")
```

Because the **print_cylinder_volume** function in example 1 doesn't accept parameters, it must be called without any arguments like this:

```
print_cylinder_volume()
```

How to Make a User-Defined Function Reusable

Because the **print_cylinder_volume** function in example 1 gets input from a user and prints its results to a terminal window, it can be used only in a program that runs when a user is present. It cannot be used in a program that runs automatically and gets input from a file or the network or a sensor. In other words, the **print_cylinder_volume** function in example 1 is not reusable in other programs. The most *reusable functions* are ones that take parameters, perform calculations, and return a result but *do not perform user input and output*.

The parameter list in a function's header contains data stored in variables that the function needs to complete its task. A *parameter* is a variable whose value comes from outside the function. One way to get input into a function is to ask the user for input by calling the built-in Python **input** function. Another way to get input into a function is through the function's parameters. Getting input through parameters is much more flexible than asking the user for input because the input through parameters can come from the user or a file on a hard drive or the network or a sensor or even another function.

Example 2 contains another version of the **print_cylinder_volume** function. This second version doesn't get the radius and height from the user. Instead, it gets input through its two parameters named *radius* and *height*.

```
# Example 2
import math
# Define a function named print_cylinder_volume.
def print_cylinder_volume(radius, height):
  """Compute and print the volume of a cylinder.
  Parameters
  radius: the radius of the cylinder
  height: the height of the cylinder
  Return: nothing
```

```
    """
    # Compute the volume of the cylinder.
    volume = math.pi * radius**2 * height
    # Print the volume of the cylinder.
    print(volume)
```

Because the second version of the **print_cylinder_volume** function accepts two parameters, it must be called with two arguments like this:

```
print_cylinder_volume(2.5, 4.1)
```

To *return* a result from a function, simply type the keyword **return** followed by whatever result you want returned to the calling function. Example 3 contains a third version of the cylinder volume function. Notice that the version in example 3 returns the volume instead of printing it, which makes the function more reusable. Notice also in example 3 that we changed the name of the function from **print_cylinder_volume** to **compute_cylinder_volume** because this version doesn't print the volume but instead returns it.

```
# Example 3
import math
# Define a function named computer_cylinder_volume.
def compute_cylinder_volume(radius, height):
    """Compute and return the volume of a cylinder.
    Parameters
    radius: the radius of the cylinder
    height: the height of the cylinder
    Return: the volume of the cylinder
    """
    # Compute the volume of the cylinder.
    volume = math.pi * radius**2 * height
    # Return the volume of the cylinder so that the
    # volume can be used somewhere else in the program.
    return volume
```

Many functions that you've used in the past such as **input**, **float**, and **round**, return a result. When a function returns a result, we usually write code to store that returned result in a variable to use later in the program like this:

```
text = input("Please enter your name: ")
```

Because the **compute_cylinder_volume** function in example 3 accepts two parameters and returns a result, it could be called like this:

```
volume = compute_cylinder_volume(2.5, 4.1)
```

The main User-Defined Function

In all previous Python programs that you wrote in CSE 110 and 111, you wrote statements that were not in a function like the simple program in example 4.

```
# Example 4
import math
# Get the radius and height from the user.
radius = float(input("Enter the radius of a cylinder: "))
height = float(input("Enter the height of a cylinder: "))
# Compute the volume of the cylinder.
volume = math.pi * radius**2 * height
# Print the volume of the cylinder.
print(f"Volume: {volume:.2f}")
```

```
> python example_4.py
Enter the radius in centimeters: 3
Enter the height in centimeters: 8
Volume: 226.19
```

In a large program, writing statements outside a function can lead to poor organization. Professional software developers write statements inside a function whenever possible. Beginning with this lesson, you will do the following in each program:

1. Write nearly all statements inside a user-defined function.
2. Write a user-defined function named **main**, which contains the beginning statements of your program.
3. Write one or more user-defined functions that have parameteters, perform calculations and other useful work, and return a result to the call point.
4. Write a call to the **main** function at the bottom of your program.

Example 5 contains the same Python program as example 4 except most of the statements are inside a user-defined function named **main**.

```python
# Example 5
import math
# Define a function named main.
def main():
    # Get the radius and height from the user.
    radius = float(input("Enter the radius of a cylinder: "))
    height = float(input("Enter the height of a cylinder: "))
    # Compute the volume of the cylinder.
    volume = math.pi * radius**2 * height
    # Print the volume of the cylinder.
    print(f"Volume: {volume:.2f}")
# Start this program by
# calling the main function.
main()
```

```
> python example_5.py
Enter the radius in centimeters: 3
Enter the height in centimeters: 8
Volume: 226.19
```

Notice the call to the **`main`** function in example 5 at line 14. Without that call to the **`main`** function, when we run the program, the program will do nothing. In all of your future programs in CSE 111 you will write a user-defined function named **`main`** and will write a call to **`main`** at the bottom of the program.

A Complete Program with User-Defined Functions

If you look closely at the code in examples 1 and 5, you will realize that both programs have the same problem, namely both the **`print_cylinder_volume`** function in example 1 and the **`main`** function in example 5 are not reusable because both of them get input from a user and print to a terminal window. A better way to write the program in examples 1 and 5 is to separate the program into two functions, one named **`main`** and one named **`compute_cylinder_volume`** as shown in example 6.

Example 6 contains a complete program with two functions, the first named **`main`** at line 4 and the second named **`compute_cylinder_volume`** at line 14. At line 10, the **`main`** function calls the **`compute_cylinder_volume`** function. Notice that the **`compute_cylinder_volume`** function gets its input through parameters and returns a result which makes this function reusable in other programs, including programs that run automatically without a user.

```python
# Example 6
import math
# Define the main function.
def main():
    # Get a radius and a height from the user.
    radius = float(input("Enter the radius of a cylinder: "))
    height = float(input("Enter the height of a cylinder: "))
    # Call the compute_cylinder_volume function and store
    # its return value in a variable to use later.
    volume = compute_cylinder_volume(radius, height)
    # Print the volume of the cylinder.
    print(f"Volume: {volume:.2f}")
# Define a function that accepts two parameters.
def compute_cylinder_volume(radius, height):
    """Compute and print the volume of a cylinder.
    Parameters
    radius: the radius of the cylinder
    height: the height of the cylinder
    Return: the volume of the cylinder
    """
    # Compute the volume of the cylinder.
    volume = math.pi * radius**2 * height
    # Return the volume of the cylinder so that the
    # volume can be used somewhere else in the program.
    # The returned result will be available wherever
    # this function was called.
    return volume
# Start this program by
# calling the main function.
main()
```

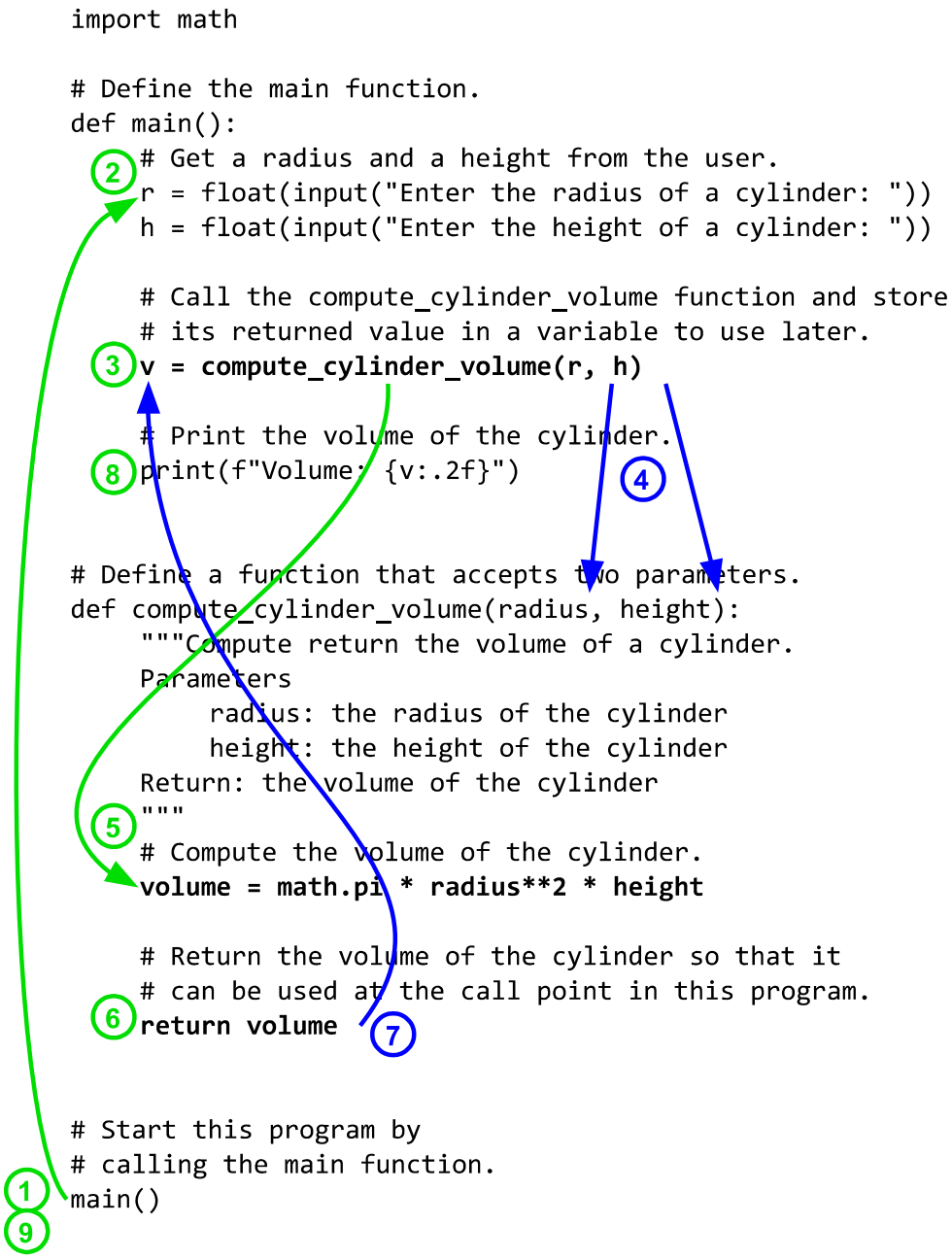
```
> python example_6.py
Enter the radius in centimeters: 3
Enter the height in centimeters: 8
Volume: 226.19
```

The most reusable functions are ones that have parameters, perform calculations, and return a result but *do not perform user input and output*. In the previous code example, there are two functions named **`main`** and **`compute_cylinder_volume`**. The **`main`** function is certainly useful in the program, but it is not reusable in other programs because it gets user input and prints the result for the user to see. The **`compute_cylinder_volume`** function is very reusable in another program because it doesn't get user input or print output. Instead, it takes two parameters, performs a

calculation, and returns a result to the calling function. The **`compute_cylinder_volume`** function is so reusable that it could be included in a library of functions that compute the area and volume of 2-D and 3-D geometric shapes.

What Happens When the Computer Calls a Function?

Some students have trouble visualizing what happens when the computer calls (executes) a function. The following diagram contains the same program as example 6. The circled numbers show the order in which the events happen in the computer. The green numbers and arrows in the diagram show the order in which the computer executes statements in the program. The blue numbers and arrows show how data flows from arguments into parameters and from a returned result to a variable.



A computer will execute the statements in the previous diagram in the following order:

A. The statement at (1) is not inside a function, so the computer executes it when the program begins. The statement at (1) is a call to the **`main`** function which causes the computer to begin executing the statements inside **`main`** at (2).

B. At (2), the computer gets two numbers from the user.

C. The statement at (3) is a call to the **`compute_cylinder_volume`** function which causes the computer to copy the values in the arguments *r* and *h* into the parameters *radius* and *height* respectively and then begin executing the statements inside the **`compute_cylinder_volume`** function at (5).

D. At (5), the computer computes the volume of a cylinder.

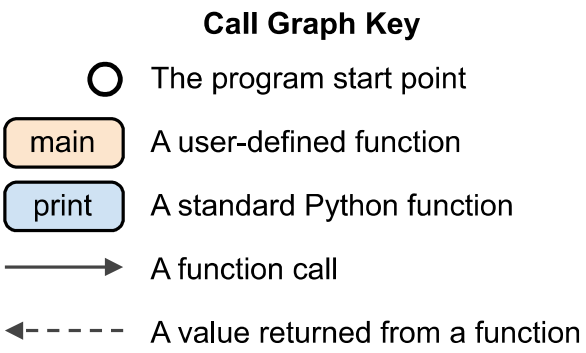
E. The statement at (6) is a return statement which causes the computer to stop executing the **`compute_cylinder_volume`** function, to return the computed volume to the call point at (3), and to resume executing statements at the call point.

F. At the call point (3), the computer stores the returned value in the variable named *v*.

G. At (8), the computer prints the value that is in the *volume* variable for the user to see. This is the last statement in the **`main`** function, so after executing it, the computer resumes executing the statements after the call point (1) to **`main`**.

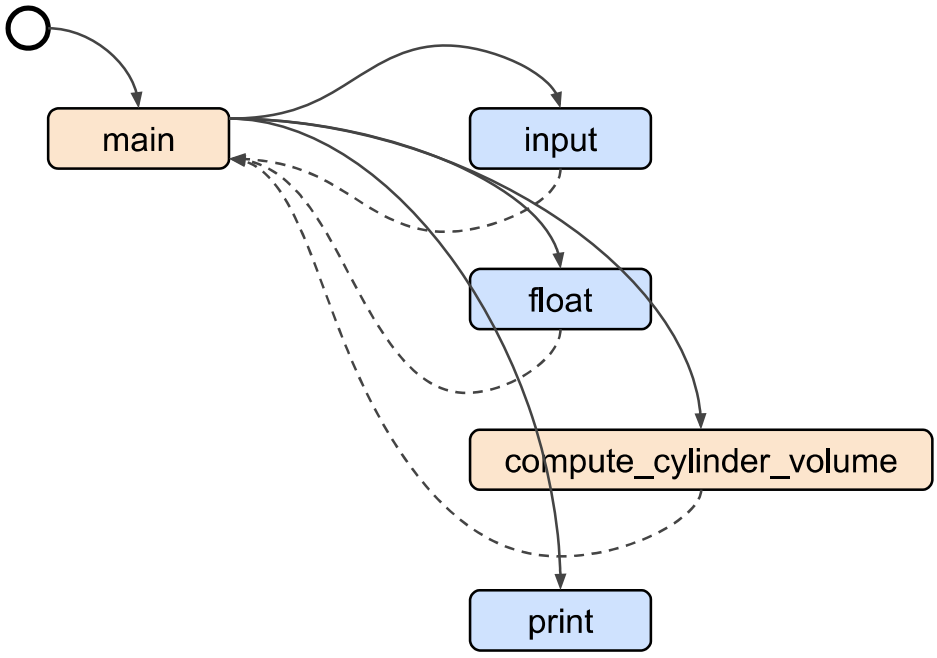
H. At (9), there are no more statements after the call to **`main`**, so the computer terminates the program.

Call Graphs



A *call graph* is a diagram that shows function calls and returns within a program. A call graph can help you visualize how a program is divided into functions. Within a call graph, the unfilled circle shows where the computer begins executing a program. A rounded rectangle represents a function. A solid arrow represents a call from one function to another function. A dashed arrow represents a value returned from a called function to the calling function.

The call graph below shows the function calls and returns for the program in example 6. From the call graph, we see that the computer begins executing the program by calling the **`main`** function. While executing the **`main`** function, the computer calls the **`input`** and **`float`** functions. Then the computer calls the **`compute_cylinder_volume`** function. Finally the computer calls the **`print`** function. In the call graph we can see that the **`main`** and **`print`** functions don't return a value. The **`print`** function prints results for the user to see, but it doesn't return anything.



Summary

A function is a group of statements that together perform one task. A user-defined function is a function written by a programmer like you. To write a user-defined function, write code that follows this template:

```
def function_name(param1, param2, … paramN):
  """documentation string"""
  statement1
  statement2
  ⋮
  statementN
  return value
```

To call a user-defined function, write code that follows this template:

```
variable_name = function_name(arg1, arg2, … argN)
```

The most reusable functions are ones that take parameters, perform calculations, and return a result but *do not perform user input and output*. All of your future programs in CSE 111 will have a user-defined function named **`main`** and will have a call to **`main`** at the bottom of the program.

It is extremely important that you can write and call functions. After watching the videos and reading this preparation content, if the concepts still seem confusing or vague to you, pray and ask Heavenly Father to help you understand the concepts. Then watch the videos and read the concepts **again**.

Checkpoint: Writing Functions

Purpose

Check your understanding of writing your own functions with parameters and then calling those functions with arguments.

Problem Statement

Many vehicle owners record the fuel efficiency of their vehicles as a way to track the health of the vehicle. If the fuel efficiency of a vehicle suddenly drops, there is probably something wrong with the engine or drive train of the vehicle. In the United States, fuel efficiency for gasoline powered vehicles is calculated as miles per gallon. In most other countries, fuel efficiency is calculated as liters per 100 kilometers.

The formula for computing fuel efficiency in miles per gallon is the following:

$$mpg = \frac{end - start}{gallons}$$

where *start* and *end* are both odometer values in miles and *gallons* is a fuel amount in U.S. gallons.

The formula for converting miles per gallon to liters per 100 kilometers is the following:

$$lp100k = \frac{235.215}{mpg}$$

Assignment

Write a Python program that asks the user for three numbers:

1. A starting odometer value in miles
2. An ending odometer value in miles
3. An amount of fuel in gallons

Your program must calculate and print fuel efficiency in both miles per gallon and liters per 100 kilometers. Your program must have three functions named as follows:

1. **`main`**
2. **`miles_per_gallon`**
3. **`lp100k_from_mpg`**

All user input and printing must be in the **`main`** function. In other words, the **`miles_per_gallon`** and **`lp100k_from_mpg`** functions must not call the the **`input`** or **`print`** functions.

Helpful Documentation

- The preparation content for the previous lesson explains how to call a function.

Steps

Copy and paste the following code into a new program named **`fuel_usage.py`**. Use the pasted code as a design as you write your program. Write code for each of the three functions.

```
def main():
    # Get an odometer value in U.S. miles from the user.
    # Get another odometer value in U.S. miles from the user.
    # Get a fuel amount in U.S. gallons from the user.
    # Call the miles_per_gallon function and store
    # the result in a variable named mpg.
    # Call the lp100k_from_mpg function to convert the
    # miles per gallon to liters per 100 kilometers and
    # store the result in a variable named lp100k.
    # Display the results for the user to see.
    pass
def miles_per_gallon(start_miles, end_miles, amount_gallons):
    """Compute and return the average number of miles
    that a vehicle traveled per gallon of fuel.
    Parameters
    start_miles: An odometer value in miles.
    end_miles: Another odometer value in miles.
    amount_gallons: A fuel amount in U.S. gallons.
    Return: Fuel efficiency in miles per gallon.
    """
    return
def lp100k_from_mpg(mpg):
```

```
    """Convert miles per gallon to liters per 100
    kilometers and return the converted value.
    Parameter mpg: A value in miles per gallon
    Return: The converted value in liters per 100km.
    """
    return
# Call the main function so that
# this program will start executing.
main()
```

Testing Procedure

Verify that your program works correctly by following each step in this testing procedure:

1. Run your program and enter the inputs shown below. Ensure that your program's output matches the output below.

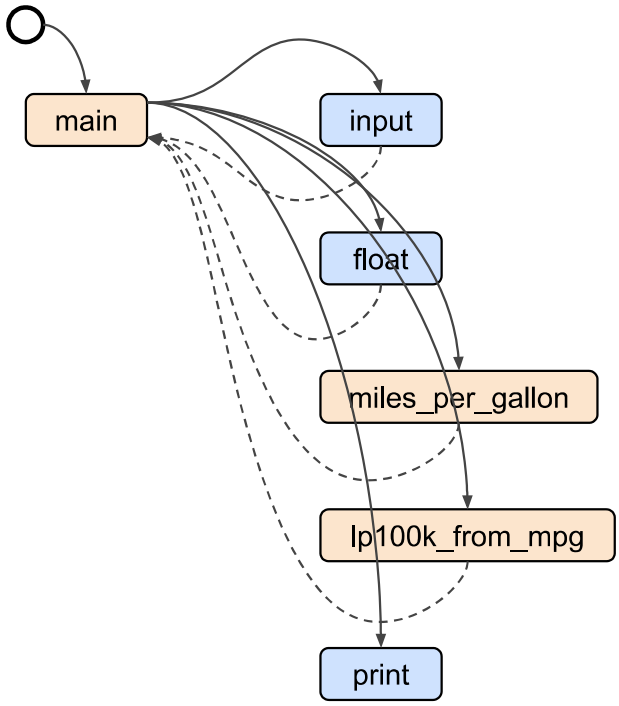
```
> python fuel_usage.py
Enter the first odometer reading (miles): 30462
Enter the second odometer reading (miles): 30810
Enter the amount of fuel used (gallons): 11.2
31.1 miles per gallon
7.57 liters per 100 kilometers
```

Sample Solution

When your program is finished, view the sample solution for this assignment to compare your solution to that one. Before looking at the sample solution, you should work to complete this checkpoint program. However, if you have worked on it for at least an hour and are still having problems, feel free to use the sample solution to help you finish your program.

Call Graph

The following call graph shows the function calls and returns in the sample solution for this assignment. From this call graph we see that the computer starts executing the sample program by calling the **`main`** function. While executing the **`main`** function, the computer calls the **`input`** and **`float`** functions. Then the computer calls the **`miles_per_gallon`** and **`lp100k_from_mpg`** functions. Finally the computer calls the **`print`** function which is the end of the program.



Ponder

After you finish this assignment, congratulate yourself because you wrote a Python program with three user-defined functions named **`main`**, **`miles_per_gallon`**, and **`lp100k_from_mpg`**. Is it important that you know how to write your own functions? Why?

Up Next

- W02 Learning Activities: Function Details (2 of 2)

Useful Links:

- Return to: Week Overview | Course Home