Learning Activity (2 of 2): Mathematical Expressions

Overview

In the previous lesson, you learned how to work with numeric data types. In this lesson, you will learn how to solve more interesting problems using numeric data types.

Once you've mastered these concepts, you will be capable of writing programs to solve even the most complicated mathematical or scientific computations, it is simply a matter of adding more variables and using larger expressions, but the principles involved are the same.

Preparation Material

Rounding

When the computer stores floating point numbers (decimals), it stores them internally in a form of binary scientific notation, and it will store them to a large number of decimal places for you.

Sometimes, you want to display numbers to the user, but you don't want to display the full number of decimal places, so you want to round it. There are various approaches to rounding numbers, depending on what you want the computer to *store* internally versus what you want it to *display* to the user. One of the simplest and most common approaches is to allow the computer to continue to store the number in a precise way, but format it nicely for the user when you display it.

Using Format Strings

To round your numbers for display purposes only (but continue to have the computer store the full value internally), you'll make use of format strings. You have been using these already, but you will now learn more options for them.

To use a format string you leave placeholders for your variables using curly braces **{}** and then the value of the variable is inserted into that spot in the string. This can be done in two different ways:

The simplest way, was recently added to python, and that is to put an **`f`** right before your string, as shown:

```
print(f"There are {car_count} cars on the road.")
```

Another approach is to not use the **`f`** before the string, but instead, use **`.format()`** after the string, as shown:

```
print("There are {} cars on the road.".format(car_count))
```

The effect of these approaches is the same, but the first approach using the "f" is a type of shorthand notation, to make things simpler and cleaner. It is important to know about the second way, because you will see it used in many code examples on the internet. Also, the second way is a more standard approach that you will see in other languages (for example, Java, C#, etc.).

For this course, you are welcome to use either approach, but most of the material will assume you are using the simpler, shortcut, **`f`** string notation.

Format String Options

There are many different options for formatting numbers using format strings. (If you are interested, you can read the official documentation on Python's Common string operations web page, but be warned that it's a little dense.)

In short, there are options for left/right/center alignment, expressing numbers in different bases (base 10 verse base 16), formatting dates in many different ways, and many other things. The most common formatting you will likely use is to define the number of decimals, define the separator for thousands (the comma), or to use scientific notations. These can be done as follows:

Defining the number of decimals to display

In a format string, you define the precision, or number of decimals to display, by putting a **`:.2f`** after the variable name (changing the 2 to whichever amount you'd like).

- The colon (**`:`**) after the variable name indicates that you are going to specify how to format it.

- The period (`.`) indicates that you are setting the precision or number of decimal places.
- The number (in this example `2`) indicates that you would like that number of decimal places to be displayed
- The `f` indicates that you want fixed-point notation.

The following shows this in action:

```
cars = 3
people = 8

people_per_car = people / cars

# Round to 1 decimal
print(f"There are {people_per_car:.1f} people in each car.")
# Output: There are 2.7 people in each car.

# Round to 2 decimals
print(f"There are {people_per_car:.2f} people in each car.")
# Output: There are 2.67 people in each car.

# Round to 3 decimals
print(f"There are {people_per_car:.3f} people in each car.")
# Output: There are 2.667 people in each car.
```

Scientific Notation

You can tell the computer to display the number in scientific notation, or "exponent" notation by using `:.3e` after your variable, where `3` defines the precision and `e` indicates that you are using exponent notation.

The following shows this in action:

```
distance = 9 * 1205 * 18

# Scientific notation with 3 digits of precision
print(f"The distance is: {distance:.3e} meters.")
# Output: The distance is: 1.952e+05 meters.

# Scientific notation with 6 digits of precision
print(f"The distance is: {distance:.6e} meters.")
# Output: The distance is: 1.952100e+05 meters.
```

Thousands Grouping

When you write numbers in code, you don't use commas to separate the groupings of digits (in other words, you don't write: 10,000,000, just 10000000). Recently, Python added a notation that lets you type using underscores such as 10_000_000.

In any case, when you want to display large numbers to the user, you may want to display it with commas or underscores. This is done by using either `:,` or `:_` after the variable name.

The following shows this in action:

```
big_number = 123456789

# Display without formatting:
print(f"The number is: {big_number}")
# Output: The number is: 123456789

# Display with "," formatting:
print(f"The number is: {big_number:,}")
# Output: The number is: 123,456,789

# Display with "_" formatting:
print(f"The number is: {big_number:_}")
# Output: The number is: 123_456_789
```

Using the Math library

One of the things that makes it possible to write large, complex programs is to make use of pre-existing code written by other people. When someone bundles up parts of their code to make it available to others, it is called a *library*. In future courses, you will learn about this in much more detail, including how to make your own libraries.

Python comes with many libraries built-in. One of these is called the "Math" library, and it contains common mathematical functions and operations. In order to use code from this library in your program, you'll need to import or include it. You do this by putting the code **`import math`** at the top of your program. For example, if you wanted to use the value of Pi included in the math library you do so as follows:

```
import math

radius = 5
area = math.pi * (radius ** 2)
```

```
print(f"The area is: {area}")
# Output: The area is: 78.53981633974483
```

As you might expect, there are lots of things available in the math library. You can see the list of Mathematical functions in the official documentation. If you are a scientist or engineer, you'll want to get familiar with a lot of these functions. A few that might be of interest to you are the following:

- **`math.ceil(value)`**—Rounds **`value`** up to the next whole number, the "ceiling."
- **`math.floor(value)`**—Rounds **`value`** down to the next whole number.
- **`math.exp(value)`**—Raises *e* to the power of **`value`**.
- **`math.sin(value)`**—Computes the trigonometry *sine* function of **`value`** in radians.

An example:

```
import math

weight = 1.65

lower = math.floor(weight)
print(lower)
# Output: 1

higher = math.ceil(weight)
print(higher)
# Output: 2
```

Good style

Finally, a note about good style. Whenever you write a program, you need to be conscious of the fact that your program will need to be understood and potentially reused or modified by others in the future. It may be used by other people on your team, or future employees of the company, or it may simply be you returning to your program after a few months, trying to remember what you had done.

In addition, writing clean, well-formatted code will help you avoid bugs and problems in your programs as you are writing them.

For these reasons, it is important to follow good practices of naming variables and formatting your code. Most companies have a strict style guide that code must follow, but in any case, it is important to write consistent clear code.

You will continue to learn more about style in this and future courses. But at this point, here are a few guidelines to keep in mind:

- Variable names should be meaningful and descriptive. Names like "cnt" instead of "count" or "a" instead of "area" will be harder to use later on.
- Variables should be in all lowercase with underscores _ separating words, such as: **`primary_song`**, **`first_name`**, or **`total_page_count`**.
- Include a space before and after operators:

```
# Proper style
area = length * width

# Bad examples:
area=length*width
area= length * width
area = length*width
area = length *width
```

- Use blank lines to separate blocks of your code that are similar. When you write an English paper, you break it up into paragraphs composed of multiple sentences that belong together. Similarly, when you write programs, you often have three or four lines together, and then a blank line before the next group of three or four. The important thing here is not the number of lines, as much as to separate code into groups that relate to one another.

Activity Instructions

Practice solving problems with variables and expressions, and displaying the results.

The Problem: Converting between different types of units

Temperature, like many other values, can be measured in different kinds of units. In this case, you might measure the value of degrees in Fahrenheit, Celsius, or Kelvin. Converting from one scale to another uses the same calculation over and over again for different input values.

Situations like this one are the perfect case for a program that can be coded to complete the steps of the calculation.

Assignment

Write a program to convert from Fahrenheit to Celsius. Display the result to one decimal place of precision.

To convert degrees in Fahrenheit to Celsius you need to subtract 32 from the Fahrenheit amount and then multiply it by the fraction 5/9.

Here is an example of the program when it runs:

```
What is the temperature in Fahrenheit? 81
The temperature in Celsius is 27.2 degrees.
```

Another example is:

```
What is the temperature in Fahrenheit? 12
The temperature in Celsius is -11.1 degrees.
```

Sample Solution

When your program is finished, please view a sample solution of this program to compare your approach to that one.

You should work to complete this checkpoint program first, without looking at the sample solution. However, if you have worked on it for at least an hour and are still having problems, you may feel free to use the sample solution to help you finish your program.

- Sample solution

Testing Procedure

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

1. Try the values in the examples on this page and ensure that your program generates the same results.
2. Verify that you are displaying the result to 1 decimal point of precision as shown.
3. Try converting 32 degrees Fahrenheit (freezing, which should be 0 degrees Celsius) and 212 degrees (boiling, which should be 100 degrees Celsius)
4. Try converting 0 and a negative number and make sure they come out as you would expect.
5. Verify that you have used good style, by checking the variable names you have used as well as the use of blank lines and whitespace around your operators.

Submission

You have now completed all of the learning activities for the week!

Make sure to:

- Return to Canvas and submit the associated quiz.

Up Next

- Check Your Understanding

Other Links:

- Return to: Week Overview | Course Home