1.5 Arithmetic Expressions in Python

In the two examples in the previous section, we used arithmetic expressions on the right-hand side of the assignment statement (equal sign). Python has its set of rules about how these expressions are to be evaluated, so that there is no ambiguity. So, far, we've simply illustrated that multiplication and division have higher precedence than addition and subtraction, as is frequently taught in grade school mathematics. Furthermore, parentheses have the highest precedence and can be used to "force" the order in which operations are evaluated as is illustrated in the this line of code that was previously shown:

```
total_price = item_price*(1+tax_rate/100)
```

In this expression, we first evaluate the contents of the parentheses before multiplying. In evaluating the contents of the parentheses, we first perform the division, since that has higher precedence than addition. Thus, for example, if tax_rate is 7, then we first take 7/100 to get .07 and then add that to 1 to get 1.07. Then, the current value of item_price is multiplied by 1.07, which is then assigned to total_price.

Python also provides three more operators for us to use:

1. **, for exponentiation.
2. //, for integer division
3. %, for modulus

The following sections describe exactly how each of these operators work, and the order of operations for each of them.

Exponentiation (**)

Many students first learn to associate the caret symbol (^) with exponentiation because that key is typically linked to exponentiation on most calculators that students use in grade school. However, in most programming languages, the caret symbol is either not defined or means something very different than exponentiation.

Some programming languages don't define exponentiation through an operator at all, but python does. The operator is defined to work for real numbers. Here are several examples of its use:

```
>>> 2 ** 3
8
>>> 3 ** 5
```

```
243
>>> 4 ** 10
1048576
>>> 25 ** .5
5.0
>>> 2 ** -3
0.125
>>> 9999 ** 0
1
>>> -5 ** -3
-0.008
>>> -5 ** 3
-125
>>> 1.6743 ** 2.3233
3.311554089370817
```

If both operands to an exponentiation operation are integers and the answer is an integer, then it will be expressed as one. If both operands are integers but the answer isn't, then the answer will be expressed as a real number with decimals. As the examples above illustrate, if an exponent b is negative, then a^b is defined as $1/a^{-b}$. (Technically, this rule is true for all exponents, not just negative ones.)

Integer division (/ /)

Python provides a second division operator, //, which performs integer division. In particular, the answer of an integer division operation is always an integer. In particular, a//b is defined as the largest number of whole times b divides into a. Here are a few examples of evaluating expressions with integer division:

```
>>> 25//4
6
>>> 13//6
2
>>> 100//4
25
>>> 99//100
0
>>> -17//18
-1
>>> -1//10000000
-1
```

```
>>> -99999999//99999999
-1
>>> -25//4
-7
>>> -13//6
-3
>>> -12//6
-2
>>> -11//6
-2
>>> 0//5
0
```

It's important to note that python deals with this operation differently than many other programming languages and different than most people's intuitive notion of integer division. In particular, when most people see -13//6, they tend to think that this is pretty close to -2, so that the answer should be -2. But, if we look at the technical definition of integer division in python, we see that -2 is greater than -13/6, which is roughly -2.166667, and the largest integer less than or equal to this value is really -3.

Incidentally, the definition of integer division in Python does not require the two numbers that are being divided to be integers. Thus, integer division operations are permissible on numbers that aren't integers. Consider the following examples:

```
>>> 6.4 // 3.1
2.0
>>> 3.4 // 3.5
0.0
>>> 9.9 // .03
330.0
>>> -45.3 // -11.2
4.0
>>> 45.3 // -11.2
-5.0
```

This feature of Python is used relatively rarely, so no further details will be given at this point.

Modulus Operator (%)

For those who have never programmed, it's likely that the modulus operator, denoted by the percent sign(%), is not familiar. In mathematics, modulus is typically defined for integers only. However in python, the modulus operator is defined for both integers and real numbers. If both operands are integers, then the answer will be an integer, otherwise, it will be a real number.

Intuitively, the modulus operator calculates the remainder in a division, while integer division calculates the quotient. Another way of thinking about modulus is that is simply calculates the leftover when dividing two numbers. This intuitive definition is correct when dealing with positive numbers in Python. Negative numbers are another story, however.

The formal definition of a % b is as follows:

a % b evaluates to a - (a // b)*b.

a // b represents the whole number of times b divides into a. Thus, we are looking for the total number of times b goes into a, and subtracting out of a, that many multiples of b, leaving the "leftover."

Here are some conventional examples of mod, using only non-negative numbers:

```
>>> 17 % 3
2
>>> 37 % 4
1
>>> 17 % 9
8
>>> 0 % 17
0
>>> 98 % 7
0
>>> 199 % 200
199
```

We see in these examples, if the first value is ever less than the second value, then the answer of the operation is simply the first value, since the second value divides into it 0 times. In the rest of the examples, we see that if we simply subtract out the appropriate number of multiples of the second number, we arrive at the answer.

With negative integers however, one must simply plug into the formal defintion instead of attempting to use intuition. Consider the following examples:

```
>>> 25 % -6
-5
>>> -25 % -6
-1
>>> -25 % 6
5
>>> -48 % 8
0
>>> -48 % -6
0
>>> -47 % 6
1
>>> -47 % 8
1
>>> -47 % -8
-7
```

The key issue that explains the first two results is that there is a different answer for the two integer divisions 25//-6 and -25//-6. The first evaluates to -5 while the second evaluates to 4. Thus, for the first, we calculate -6 * -5 = 30, and need to subtract 5 to obtain 25. For the second, we calculate -6 * 4 = -24, and need to subtract 1 to obtain -25.

See if you can properly apply the definition given to explain each of the other answers shown above.

In Python, the modulus operator is also defined for real numbers, using the same definition as the one shown. Here are a few examples of its use:

```
>>> 6.4 % 3.1
0.20000000000000018
>>> 3.4 % 3.5
3.4
>>> 9.9 % .03
7.216449660063518e-16
>>> -45.3 % -11.2
-0.5
>>> 45.3 % -11.2
-10.7
```

If we take a look at the first and third examples, we see the answer that python gives is slightly different than we expect. We expect the first to be 0.2 exactly and the third to be 0. Unfortunately, many real numbers are not stored perfectly in the computer. (This is true in all computer languages.) Thus, in calculations involving real numbers, occasionally there are some slight round-off errors. In this case, the digits 1 and 8 way at the right end of the number represent a round-off error. In the third case, the last part of it, e-16, simply means to multiply the previously shown number by 10^{-16}, so the entire part printed represents the round-off error, which is still tiny, since it's less than 10^{-15}. As long as we don't need extreme precision in our calculations, we can live with the slight errors produced by real number calculations on a typical computer. The more complicated calculations are, the greater the possible error could cascade to be. But, for our purposes, we'll simply assume that our real number answers are "close enough" for our purposes. (There are a variety of advanced techniques that help programmers contend with round off error of real number calculations.)

1.6 Reading User Input in Python

input statement

Python makes reading input from the user very easy. In particular, Python makes sure that there is always a prompt (a print) for the user to enter some information. Consider the following example entered

```
>>> name = input("What is your name?\n")
What is your name?
Simone
>>> print("Please to meet you ", name, ".", sep="")
Please to meet you Simone.
```

In this way, instead of the print always printing out the same name, it will print out whatever name the user entered. The key is that the input statement read in whatever the user entered, and then the assignment statement, with the equal sign, assigned this value to the variable name. Then, we were free to use name as we pleased, knowing that it stored the value the user entered.

Our previous programs that calculated the price of an item with tax and the area of a square, were limited because they always calculated the same price and area. Our program would be much more powerful if we allowed the user to enter the appropriate values so that our program could calculate the information THEY are interested in (as opposed to the same value every time.)