Homework - Expressions, Numbers, and Type Conversions

There should be 2 Python error messages when doing this assignment

Learning Outcomes

- Understand the difference between inter and floating-point data types
- Explain the order of operations for numerical expressions
- Demonstrate basic string indexing

Rubric (One Hundred Points)

4 points for turning in the document

16 points per every answered question (6)

Data Types and Numbers

When we have been using variables, we used something that looks like this:

```
num = 5
print(num)
```

As we have seen, once a variable is defined it points to an object that has a **type**. You can see a complete list of data types in the textbook. We know we can change the values, but does the type mean that you must always use the same type when you assign new values to the variables? The answer is no. Python is flexible and does not restrict a variable to being one type through its entire life.

Let us look at some of the types...

Integers and Floating-Point Numbers

Integers are whole numbers only (e.g., 1, 2, or 3 - no decimals or fractional parts). These are easy to understand. However, many things are measured with fractional numbers (like money). **Floating-point** numbers refer to any number with a decimal point. Why are these two distinct types? To show why, try the following Python code:

```
print(0.1+0.1)
print(0.1+0.2)
```

QUESTION 1. Is it what you expected? What did you expect?

Yes, this was what I expected as we did a similar exercise in class.

QUESTION 2. Before reading further, can you make a guess as to why you got what you did?

The reason is how the computer has to guess as to the actual number is because it cannot be represented properly in the system due to there being infinite numbers between each whole number.

This happens because of the way computers represent numbers internally and is not particular to Python. We are used to working in powers of ten, where one tenth plus two tenths is just three tenths, but computers work in powers of two. So, the computer must represent 0.1 in a power of two, and then 0.2 as a power of two, and express their sum as a power of two. There is no exact representation for 0.3 in powers of two, and we see that in the answer to 0.1+0.2. Computers use a lot of precision (i.e., places after the decimal point) to minimize this problem and Python tries to hide this kind of stuff when possible. Do not worry about it for now; just do not be surprised by it and know that we will learn to clean up our results a little later.

Depending on the version of Python and your host computer, you may see another floating-point number format. Really small and big numbers are hard to print out and even harder to read unless you use shorthand. Python can print floating point numbers in **scientific notation** format. For example, if you have a number 300 this can also be written as:

3×10^{2}

There is no superscript in Python code, so you would write this as ***3e+2***. The **e** in the number tells Python you are using scientific notation.

Arithmetic Expressions

When you make a Python **expression**, you can do all the basic arithmetic operations with numbers like addition and subtraction using the standard plus and minus symbols. Multiplication uses the asterisk, and division uses a forward slash. Exponents use two asterisks.

Try the following:

```
print(3+2)
print(3-2)
print(3*2)
print(3**2)
```

There is a potential side effect of some mathematical operators to note. If you were to try the expression:

```
print(3/2)
```

QUESTION 3. What do you think you would get as the answer?

1.5

Try this expression now:

```
x=4/2
print(x)
print(type(x))
```

QUESTION 4. Why do you think you got what you did?

I think that the computer is not actually dividing but multiplying by a factor and to do so it would have to multiply this by a floating point so it had to convert it to a floating point.

Order of Operations

QUESTION 5. What do you think the following generates (do not try it before guessing)?

```
result = 2+3*4
print(result)
```

It would give you 14.

Now go ahead and try it in Python:

```
result = 2+3*4
print(result)
```

This is due to the order of operations in Python. There are specific rules for how Python will evaluate mathematical expressions which involve giving certain operators a higher priority over other operators (see the lecture slides for this week). You can use the PEMDAS acronym to remember the order, but Python will go through the entire expression, left to right and look for all the highest order operators and execute those, then it will do the next highest, etc. This is why the expression ***2+3*4*** returns the value it does. Python will execute the ***3*4*** part first, and then add two to that.

You can tell Python how you want the expression evaluated by using parentheses. It is always better to use parentheses to force the order of operations because it is easier to read, and it forces Python to do the expression the way you want. Try:

```
my_order = (2+3)*4
print(my_order)
```

Strings

Strings are containers which hold zero or more characters (i.e., they can be any size). They are **ordered containers** because the characters always occur in a specific order (i.e., you do not get random characters back when you access the string). Strings are enclosed by either

single or double quotes in Python. You can use either one when making strings (as long as you start and end a string with same type of quote).

```
my_str = "This is a string."
my_str2 = 'This is also a string.'
print(my_str)
print(type(my_str))
print(my_str2)
```

Since strings are **ordered**, you can get a specific character from a string by using an **index**. An index tells Python which one of the characters you want. Indices are numeric and you use square brackets to specify the index. Try this example:

```
print(my_str[1])
```

In Python, whenever you want to index into any data type, you always put the index in square brackets. Now try the following to make a change to the string:

```
my_str[1]='q'
print(my_str)
```

QUESTION 6. Why do you think that did not work (HINT: we discussed this concept about objects last week)?

The object is immutable as it is a string and not like a list where you can change different parts of it

Data Type Conversions

One of the convenient things in Python is that you can easily convert data types to other data types. There are built-in conversion functions (e.g., ***int()***, ***str()***, ***float()***) to do this. Try the following:

```
i = 9
print(i)
f=float(i)
print(f)
s=str(f)
print(s)
```

This is handy when you are using something like the ***input()*** function, which always returns a **string** type. If you want a number and need to do math, you need to convert the string input into the numeric data type that you want. The important thing to remember about using explicit conversions is that whatever you are trying to convert must make sense as the new type. For example, if you convert a string "123" to an integer value, which makes sense, while trying to convert a string "xyz" does not make sense as an integer value. Try the following code:

```
print(int("abc"))
```