Fundamentals: Expressions and Assignment

A typical Python program is made up of one or more ***statements***, which are ***executed*** by a ***Python shell*** for their ***side effects***—e.g, to input some data, write to a file, display values in the shell, etc.

There are many kinds of Python statements. But the most fundamental is ***assignment***. An assignment statement has the following general form, where `var` denotes a ***variable*** and `exp` denotes an ***expression***:

`var = exp` (Read as: '*`var` is assigned `exp`*' or '*`var` gets `exp`*'.)

To execute the assignment, the shell first evaluates `exp` and creates an ***object*** to represent the ***value*** of `exp`. The shell then associates `var` with this object. After executing the assignment, `var` can be used in other expressions to stand for this value.

An object is essentially a representation in computer memory of a real-world value, such as an integer or a real number. Every object has a ***type***. The type determines how the expression's value is represented and how it can be used. You can find the type of the object Python's `type` function.

This exercise explores these concepts in more detail. It also requires you to experiment in the shell and to load and run a Python program.

Part (a): (Expressions and types) A ***literal*** is an expression that stands for a fixed value. The table below illustrates four kinds of literals, their types, and the values that they stand for.

Literal	**Type**	**Value**
`18  -10000`	`int`	The integers 18 and –10,000
`3.5  -3e-4`	`float`	The decimal numbers 3.5 and –0.0003
`'THIS IS AN EX-PARROT!!'`	`str`	The sequence of 22 characters between the single quotes (including spaces and punctuation)
`"No, 'e's uh,... resting."`	`str`	The sequence of 24 characters between the double quotes (including spaces and punctuation)
`"""I never!` `Yes you did!"""`	`str`	The sequence of 21 characters between the triple quotes (including spaces, punctuation, and the '***new-line character***')
`True  False`	`bool`	The Boolean values true and false

Once a variable has been assigned a value, the variable can be used as an expression; the variable stands for the value that was last assigned to it. This value also determines the variable's type. For example, executing the assignment `x = 5` associates the variable `x` with the integer value 5. After this assignment, entering the expression `x` into the shell displays a 5; moreover, after this assignment, entering `type(x)` into the shell displays `int`.

To form more complex expressions, you can apply operators and functions to other expressions, called ***arguments***. For example, after the assignment `x = 5`, entering the expression `x + 2` into the shell, causes the shell to display the value 7. In the expression `x + 2`, the integer addition operator (+) is applied to two arguments (`x` and 2).

With a partner, bring up Spyder. Press the 'Variable explorer' tab in the top right pane to show the variable explorer window and the 'iPython console' tab in the bottom right pane to show the iPython shell window. Follow the instructions below; if you or your partner are uncertain of the answer to any question or have any other questions, put a pink sticky note on your monitor.

1. In the shell, enter `1.5e3`. Q: How does the shell display the value of the literal `1.5e3`?

2. Enter a `float` with at least 20 digits, e.g., `150000000000000000000.0`.
   Q: How does the shell display the value of the literal you entered?

3. Enter a floating-point number that differs from the one in step 2 only in the least significant digit, e.g, `150000000000000000000.8`.
   Q: What do these experiments in the shell tell you about the representation and display of floating-point values?

4. Enter the identifier `x`.
   Q: Why does the shell display an error?

5. Enter the assignment `x = 1.5e3`. Q: Why doesn't the shell display anything?

   Q: What side effect was produced by executing the assignment and how does Spyder show this side effect?

   Q: What will the shell display if you now enter `x`?

   Q: What will the shell display if you now enter `type(x)`?

6. Next enter the multi-assignment: `x, y, z = 60.0, 23.5, 0.0`
   Q: What side effect occurred and how does Spyder show this side effect?

   Q: What will the shell display if you now enter `x * z` and what is the type of `x * z`?

   Q: What will the shell display if you now enter `type(x * z)`?

   Q: In the expression `x * z`, what is the operator and what are the arguments?

7. In this step, you will practice creating a program file using Spyder and running it in the shell.

   First, find the name of the ***working directory*** in the text box in the tool bar at the top of the Spyder workspace. Press the folder icon to the right of this text box. Navigate to your Desktop or Documents folder and create a folder for the CTL and then, within that, a folder for week 1. Then press the 'Choose' button. (Put a pink sticky note on your monitor if you need help.)

8. From the navigation menu at the top of the Spyder workspace, select '`File => New File...`' This will create a new '`Untitled`' sheet in the editor it (left pane) with some boilerplate comments in. Next, select '`File => Save as...`', enter a name for the file (e.g., '`firstProgram`'), and press the '`Save`' button. This saves the contents of the editor sheet in a file with the name you entered and a '`.py`' extension in the folder you selected. It also changes the name in the tab for the editor sheet to the name you entered. Also, now, every time you execute the program, Spyder will first save it to this file.

   Next, copy the following multi-assignments into this editor sheet (below the comments). Below that, add the following multi-assignments (each on a new line):

```
x, y, z = 0.0, 23.5, 0.0
a, b = 8, 19
s, t = 'ta', 'da'
```

   The editor sheet should now contain a 3-line program containing 7 assignments.

   Q: What will the shell display if you now enter the expression `x` into the shell? Is this what you expected?

   Q: Based on that experiment, what do you think the shell will display if you enter the expression `a` into the shell?

9. To run the program, press the left-most green arrow in the toolbar (the biggest green arrow).

   Q: What affect does running this program have?

   Q: What will the shell display if you now enter `a * b` ?

10. To correctly form expressions that use functions and operators, you need to know the Python ***typing rules***. Typing rules indicate how many and what types of arguments a function or operator can be applied to and the types of the values it returns. In this step, you will experiment in the shell to discover some typing rules for a few useful Python operators and functions. To keep it simple, we will consider only three types: `int`, `float`, and `str`

    Some examples to get you started: The result of step 6 suggests that multiplying two `float` values returns a `float` value. This is expressed as the typing rule:
    `float * float –> float`

    Similarly, the result of step 9 suggests that multiplying two `int` values returns an `int`, i.e.:
    `int * int –> int`

    In contrast, entering `s * s` into the shell produces a error (try it), expressed by the rule:
    `str  * str –> ERROR`

    Beside each expression below, write the typing rule suggested by entering the expression into the shell:

    `a * x`

    `x * a`

    `b * s`

    `t * a`

    `x * s`

11. Other important operators are shown in the following table. Run experiments to determine the typing rules for each. Consider arguments of types `int`, `float`, and `str`.

Start	Operator		Rules
	`-v`	negative	
	`v + w`	addition	
	`v - w`	subtraction	
	`v / w`	division	

Start	Operator		Rules
	`v // w`	quotient	
	`v % w`	remainder	
	`v ** w`	power	

Part (b): To correctly form expressions that contain many operators, you need to know the ***precedence*** of operators and how operators ***associate***. The following table shows the precedence of the Python operators that we've used so far, from highest (power) to lowest (addition and subtraction):

Operator	Description
`x**y`	power
`-x, +x`	negative and positive
`x*y, x/y, x//y, x%y`	multiplication, division, quotient, and remainder
`x+y, x-y`	addition and subtraction

Most Python operators at the same precedence level associate from left to right.
The one exception is the power operator, which associates from right to left.

For example:

1 – 3 * 2 + 7

5 – 3 – 1

2 ** 3 ** 2

Assume the following assignments were previously entered in the shell: `A = 2` and `B = 3`
Fill in the table with the value of each expression (***do these on paper!***).

Expression	Value
`A + B // 2`	
`- B - A + 2 * A`	
`B * A / A * B`	
`10 ** B ** A * -0.5`	

Expression	Value
`(A + B) // 2`	
`(- (B - A) + 2 ) * A`	
`B * A / (A * B)`	
`(10 ** B) ** (A * -0.5)`	

After filling in the table, download the file `precedence.py` from this week's '`Artifacts`' section on the CTL website; put it in the same folder that you navigated to in Step 7 of Part a. (Press the link in the '`Artifacts`' section and then select '`File => Save page as…`' in your browser and navigate to the folder).

Open the file in Spyder (use '`File => Open`'). Run it to check your answers.

Part (c): You can also use functions in forming expressions. To call a function, you enter the function name and then any arguments (function inputs) enclosed in parentheses and separated by commas. For example, executing `help(round)` calls the built-in `help` function on the argument `round`.

Enter `help(round)` into the shell. Based on what it displays and additional experiments, try to answer the following questions.

1. What is `round`?

2. How many arguments are needed to call `round`?

3. What is 1/3 to 2 digits of accuracy?

4. What is the square root of 2 to 4 digits of accuracy? (Hint: Recall that the square root of a number *n* is the same as *n* raised to the ½ power.)

5. What integer is "closest" to the cube root of 755?

For each type, Python provides a ***type constructor***, which is a function for creating values of that type; the type constructor name is the same as the type name. Test this out in the shell by executing expressions such as the following.

```
int(2.9999e2)

float(3 * 75)

float(2.9999e2)

str(3)

str(2.9999e2)

str(3/10**15)
```

Run some tests to determine answers to the following questions:

1. When does the `int` function create an `int` from a string (`str`) argument and when does it produce an error?

2. When does the `float` function create a `float` from a string (`str`) argument and when does it produce an error?