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, change the values of one or more variables, 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:

\[ \text{var} = \text{exp} \]

(It is read as ‘\text{var} is assigned \text{exp}’ or ‘\text{var} becomes \text{exp}’.)

To execute this assignment, the shell first executes exp, which creates an in-memory representation for the value of exp, called an object. The shell then associates var with this object. After executing the assignment, var can be used in other expressions to stand for this value.

Every expression has a (data) type. The type determines how the expression’s value is represented and how it can be used.

This exercise explores these concepts in more detail. It also requires you to experiment in the Python 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.

<table>
<thead>
<tr>
<th>Literal</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>int</td>
<td>The integers 18 and -10,000</td>
</tr>
<tr>
<td>-10000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.5</td>
<td>float</td>
<td>The decimal numbers 3.5 and -0.0003</td>
</tr>
<tr>
<td>-3e-4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>'THIS IS AN EX–PARROT!!'</td>
<td>str</td>
<td>The sequence of 22 characters between the single quotes (including spaces and punctuation)</td>
</tr>
<tr>
<td>&quot;No, 'e's uh,... resting.&quot;</td>
<td>str</td>
<td>The sequence of 24 characters between the double quotes (including spaces and punctuation)</td>
</tr>
<tr>
<td>&quot;&quot;&quot;I never! Yes you did!&quot;&quot;&quot;&quot;</td>
<td>str</td>
<td>The sequence of 21 characters between the triple quotes (including spaces, punctuation, and the ‘return character’)</td>
</tr>
<tr>
<td>True</td>
<td>bool</td>
<td>The Boolean values true and false</td>
</tr>
<tr>
<td>False</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Once a variable has been assigned a value, the variable can be used as an expression; in this case, 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, the type of $x$ is int.

To form more complex expressions, you can apply operators and functions to other expressions, called arguments. For example, after the assignment $x = 5$, you can enter the expression $x + 2$ into the shell, causing 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, raise your hand for help.

1. In the shell, enter $1.5e3$ (i.e., position your cursor at the end of the last line in the shell window; then type $1.5e3$; finally press the enter-key). Q: How does the shell display the value of the literal $1.5e3$?

2. Next, enter a float with at least 20 digits, e.g., $1500000000000000000000.0$. Q: How does the shell display the value of the literal you entered?

3. Next, enter a floating-point number that differs from the one in step 2 only in the least significant digit, e.g., $1500000000000000000000.8$. Q: What do these experiments in the shell tell you about the representation and display of floating-point values?

4. Next, enter the identifier $x$ (i.e., position your cursor at the end of the last line in the shell window; then type the character $x$; finally press the enter-key). Q: Why does the shell display an error?

5. Next, 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 $x$ and what is the type of $x$?
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: In this expression, 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. If you prefer to keep your Python files in some other folder (e.g., a Dropbox folder for backup or your Desktop for easy access), navigate to that folder. In any case, once the folder is selected, create a new folder for today’s files—for example, CTLwk1—and press the ‘Choose’ button.

8. Next, from the navigation menu along the top of the Spyder workspace, select ‘File => New File...’ This will create a new ‘Untitled’ sheet in the editor with some boilerplate comments in it (left pane). Next, select ‘File => Save as...’, enter a name for the file (e.g., ‘nothingMuch’), and press the ‘Save’ button. This should save the contents of the editor sheet in a file with the name you entered and a ‘.py’ extension in the folder you selected. It should also change 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 multi-assignment from step 6 into this editor sheet (below the comments). Below that, add the following multi-assignments (each on a new line):

\[
\begin{align*}
a, b &= 8, 19 \\
s, t &= 'ta', 'da'
\end{align*}
\]

The editor sheet should now contain a 3-line program containing 5 assignments.

Q: What will the shell display if you now enter the expression \( x \) into the shell?

Q: In contrast, what will the shell display if you now enter the expression \( a \) into the shell? Why?

9. Next, ‘Run’ the program by pressing 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 \rightarrow float

Similarly, the result of step 8 suggests that multiplying two int values returns an int, i.e.:
int * int \rightarrow int

In contrast, entering str * str into the shell produces a error (try it), expressed by the rule:
str * str \rightarrow 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 that you need to learn are shown in the following table (which continues on the next page). We will assign you one or two rows of the table to discover the typing rules for. If there is time left after you have found the type rules for the rows you were assigned, you can move on to other rows. But do the rows you were assigned first, since we’ll stop after 5-10 minutes to discuss your findings.

<table>
<thead>
<tr>
<th>Start</th>
<th>Operator</th>
<th>Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>+v</td>
<td>positive</td>
<td></td>
</tr>
<tr>
<td>-v</td>
<td>negative</td>
<td></td>
</tr>
<tr>
<td>Start</td>
<td>Operator</td>
<td>Rules</td>
</tr>
<tr>
<td>-------</td>
<td>----------</td>
<td>-------</td>
</tr>
<tr>
<td>v + w</td>
<td>addition</td>
<td></td>
</tr>
<tr>
<td>v - w</td>
<td>subtraction</td>
<td></td>
</tr>
<tr>
<td>v / w</td>
<td>division</td>
<td></td>
</tr>
<tr>
<td>v // w</td>
<td>quotient</td>
<td></td>
</tr>
<tr>
<td>v % w</td>
<td>remainder</td>
<td></td>
</tr>
<tr>
<td>v ** w</td>
<td>power</td>
<td></td>
</tr>
</tbody>
</table>
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):

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x**y</code></td>
<td>power</td>
</tr>
<tr>
<td><code>-x, +x</code></td>
<td>negative and positive</td>
</tr>
<tr>
<td><code>x*y, x/y, x//y, x%y</code></td>
<td>multiplication, division, quotient, and remainder</td>
</tr>
<tr>
<td><code>x+y, x-y</code></td>
<td>addition and subtraction</td>
</tr>
</tbody>
</table>

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:

\[
1 - 3 \times 2 + 7
\]

\[
5 - 3 - 1
\]

\[
2 ** 3 ** 2
\]

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!).

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A + B // 2 )</td>
<td></td>
</tr>
<tr>
<td>( -B - A + 2 * A )</td>
<td></td>
</tr>
<tr>
<td>( B * A / A * B )</td>
<td></td>
</tr>
<tr>
<td>( 10 ** B ** A * -0.5 )</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( (A + B) // 2 )</td>
<td></td>
</tr>
<tr>
<td>( -(B - A) + 2 ) * A</td>
<td></td>
</tr>
<tr>
<td>( B * A / (A * B) )</td>
<td></td>
</tr>
<tr>
<td>( (10 ** B) ** (A * -0.5) )</td>
<td></td>
</tr>
</tbody>
</table>
Then, download the file precedence.py from this week’s ‘Artifacts’ section on the CTL website into the same folder that Spyder is currently working in (press the precedence.py-link and then select ‘File => Save page as...’ in your browser and navigate to the folder). Open the file in Spyder (using ‘File => Open’ in Spyder). Run it (press the big green arrow in the Spyder toolbar) to check your answers.

**Part (c):** You can also use functions in forming expressions, provided that the functions return values. This last part of this exercise will illustrate the use of some *type construction* functions and of the *input* and *print* functions.

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

```python
int(3)
int(2.9999e2)
float(3)
float(2.9999e2)
str(3)
str(2.9999e2)
```

What can you infer about how Python constructs an int, float, or str from a numeric argument (i.e., from an int or a float)?

Now run some tests to determine answers to the following questions:

1. Under what circumstances can the int function construct an int from a string (str) argument?

2. Under what circumstances can the float function construct a float from a string (str) argument?

3. Under what circumstances can the str function construct a string from a string argument?

The Python input function allows a *user* to provide input data to a program. The input function needs a single argument of type str, commonly called a *prompt* (because it is used to prompt the user for some input). An input expression has the following form, where str_exp denotes an expression of type string:

---

1 A programmer typically writes a program for other people to use. Those other people are called “users” (since they use the program). When you are writing a program, you are the “programmer.” When you are executing it, you are the “user.”
Python executes this expression in three steps:
1. It displays, in the shell, the string object produced by executing `str_exp`.
2. It waits for the user to type any number of characters, followed by the ‘enter’-key.
3. It creates and returns the string containing the sequence of characters typed by the user (every character up to but not including the ‘enter’-key).

Test the input expression in the shell by entering the following:
```
input('Enter a number followed by the "enter" key: ')
```

Q: What happens if you enter a non-numeric value at the prompt instead of a number?

Next, add the input expression to your program from part (a) and then rerun the program.

Q: Why do you think the shell doesn’t display anything when it executes the input expression?

To use the string returned by an input expression, the string either has to be assigned to a variable or passed as the argument to a function.

Modify your program so that the input expression is assigned to a variable:
```
num = input('Enter a number followed by the "enter" key: ')
```

Rerun the program and enter a number at the prompt, as requested.

Q: If you enter the expression `2*num` into the shell, what will the shell display now?

Modify your program so that, after executing the program, `2*num` will display the number that is twice that entered by a user. (Hint: use the `float` construction function.)

Finally, you will use the `print` function to make the program display the first 5 multiples of the number entered by a user, along with a message indicating what is being printed.

The `print` function cannot be used in forming expressions. It is executed only for its side effect (displaying some values in the shell or writing them to a file). It can be applied to any number of arguments (including no arguments). The arguments can be of any types. A comma must separate multiple arguments.

To execute the `print` function, the shell applies the `str` function to each argument in turn and displays the strings so produced, all on the same line and separated by a single ‘space’-character.

To finish off your program, add a statement such as the following to the end of the program:
```
print('The first 5 multiples of', num, 'are:', 0*num, num, 2*num, 3*num, 4*num)
```

Test your program 2-3 times.

Extra for experts: Modify the program to print the numbers with a single digit after the decimal point. (Hint: check out the `round` function.)