Carleton University
Department of Systems and Computer Engineering
SYSC 1005 - Introduction to Software Development - Fall 2018

Lab 1 - Using Software Experiments to Learn about Python's Datatypes

Objectives

- To learn how to use the Python shell to perform software experiments.
- To learn more about the Python datatypes that represent integers and real numbers.

Attendance/Sign-out

When you have finished all the exercises, call a TA, who will record that you've finished the lab. For those who don't finish early, the TAs will record how much you've completed, starting at about 15 minutes before the end of the lab period. Finish the remaining exercises on your own time, before next week's lab.

Overview

In this lab, you'll use the Python shell to explore how Python supports calculations with integers and real numbers. Our approach will use *directed experimentation*; that is, experimentation to learn something that wasn't known before.

For some exercises, you'll be asked to type expressions in the Python shell window, record the results displayed by Python, and then draw conclusions.

For other exercises, you will be asked to devise one or more short experiments. You'll record your experiments (i.e., write the Python expressions that you typed and the results displayed by Python), then write one or two sentences that summarize what you learned.

Getting Started

Launch Wing 101. One of the tabbed windows is titled Python Shell. When the shell starts, Python prints a message in this window. The first line should contain "v3.7.0", which indicates that Wing is running Python version 3.7. If another version is running, ask a TA for help to reconfigure Wing to run Python 3.7.

For this lab, you'll do all of your work in the shell. You won't be using an editor to write complete Python scripts (programs), so you don't need to open a new editor window.

Exercise 1: The shell displays a prompt (`>>>`) when it is ready for you to type a command. After the prompt, type this *expression* (don't type any spaces between the numbers and the plus sign):

```
>>> 1+2
```

The integers 1 and 2 are the expression's *operands* and the `+` is the *addition operator*.

Now press the `Enter` key. This "tells" Python to *read* this expression, verify that it is syntactically correct, then *evaluate* the expression (perform the addition operation on the two operands) and *print* (display) the result in the shell window.

On the ruled line, write the value that is displayed:

```
>>> 1+2
```

_____

Exercise 2: Are spaces in expressions significant? This time, type exactly one space between the operands and the operator, like this:

```
>>> 1 + 2
```

Remember to press `Enter` when you are ready for Python to process the expression.

On the ruled line, write the value that is displayed:

```
>>> 1 + 2
```

_____

Repeat the experiment, but this time, type several spaces between the operands and the operator:

```
>>> 1      +        2
```

Write the value that is displayed:

```
>>> 1      +        2
```

_____

What do you conclude about the significance of spaces? Does the number of spaces between the operands and operator affect the results? Record your conclusions here:

_____

Exercise 3: Can expressions be split across multiple lines? For example, instead of typing `1 + 2` on a single line, can we type `1 +` on one line and `2` on the next? In other words, is

```
1 +
2
```

a valid Python expression? Try typing the first line:

```
>>> 1 +
```

Record what you observed. Did the Python shell wait for you to type the rest of the expression (the `2`)?

_____

_____

Repeat the experiment, but this time, type a backslash, `\`, at the end of the line:

```
>>> 1 + \
```

This is what you should see:

```
>>> 1 + \
...
```

(If you don't see the three dots, did you remember to press `Enter` after typing the backslash?) The three dots are the *secondary shell prompt*; they indicate that the shell is waiting for you to finish typing the expression. Complete the expression by typing:

```
2
```

In other words, you should see this. Record the result:

```
>>> 1 + \
... 2
```

_____

Can the expression be split before the operator? Try this experiment:

```
>>> 1 \
... + 2
```

Record the result:

```
>>> 1 \
... + 2
```

_____

Exercise 4: What about other arithmetic operations on integers? Type these expressions and record the results.

```
>>> 4 - 1
```

_____

```
>>> 2 * 3
```

_____

```
>>> 6 / 2
```

_____

```
>>> 7 / 2
```

_____

```
>>> 10 / 3
```

_____

```
>>> 10 / 6
```

_____

It appears that, for integer values, `-` is the subtraction operator, `*` is the multiplication operator, and `/` is the division operator. Notice that division of two integers always yields a real number, even when the result could be represented by integer (note the result obtained when 6 is divided by 2).

Exercise 5: Does Python support more complicated expressions made up of several operands and operators? Type these expressions and record the results:

```
>>> 1 + 2 + 3
```

_____

```
>>> 5 - 1 - 1 - 1
```

_____

```
>>> 2 * 2 * 3
```

_____

```
>>> 12 / 3 / 2
```

_____

Write your conclusions here:

_____

_____

Exercise 6: How about mathematical expressions that we'd expect to evaluate to negative numbers? Type these expressions and record the results:

```
>>> 4 - 6
```

_____

```
>>> 2 * -3
```

_____

```
>>> -6 / 2
```

_____

```
>>> -7 / 2
```

_____

```
>>> 7 / -2
```

_____

Write your conclusions here:

_____

_____

Exercise 7: We can easily show that we can mix operators in an expression, but then we have to consider the order in which the operations are performed.

Consider these two expressions: `1 + 2 * 3` and `2 * 3 + 1`.

If the * operator has the same *precedence* as the + operator, the expressions will be evaluated left-to-right; that is, when the first expression is evaluated, the addition will be performed before the multiplication, and when the second expression is evaluated, the multiplication will be performed before the addition.

On the other hand, if the * operator has higher precedence than the + operator, the multiplication will always be performed before the addition, and both expressions will evaluate to the same value.

Type the expressions and record the results::

```
>>> 1 + 2 * 3
```

_____

```
>>> 2 * 3 + 1
```

_____

What do you conclude about the precedence of the * operator relative to the + operator?

_____

_____

Exercise 8: We can use parentheses (round brackets) to change the order of evaluation. Type the following expressions and record the results.

```
>>> (1 + 2) * 3
```

_____

```
>>> 2 * (3 + 1)
```

_____

Compare these results to the ones you obtained in Exercise 7.

_____

_____

_____

_____

Exercise 9: We hypothesize that subtraction has lower precedence than multiplication, which means that multiplication will always be performed before subtraction when an expression is evaluated. We also hypothesize that subtraction has the same precedence as addition. This means that an expression containing (only) the `-` and `+` operators will always be evaluated left to right.

Design a few experiments to help you determine if these hypotheses are true. On the ruled lines, write the expressions you type and the results displayed by Python (use a separate page if you need more room). Are these hypotheses correct? Record your conclusions.

Exercise 10: Can Python expressions have superfluous parentheses? Try these experiments (as always, record the results Python displays):

```
>>> (1 + 2 + 3)
```

_____

```
>>> (1) + 2
```

_____

```
>>> (((1 + 2) + 3) + 4) + 5
```

_____

```
>>> ((((1))))
```

_____

What do you conclude?

_____

_____

Exercise 11: What about missing, mismatched or incorrectly placed parentheses? Try these experiments:

```
>>> 1 + 2)
```

_____

```
>>> 1) + (2
```

_____

```
>>> (1+)2
```

_____

What do you conclude?

_____

_____

Exercise 12: What happens if you type an expression that has a left-hand parenthesis without the corresponding right-hand parenthesis? Try this:

```
>>> (1 + 2
```

This is what you should see:

```
>>> (1 + 2
...
```

The three dots are the *secondary shell prompt*; they indicate that the shell is waiting for you to finish typing the expression. Complete the expression by typing:

```
 + 3)
```

In other words, you should see this. Record the result:

```
>>> (1 + 2
... + 3)
```

_____

It looks like we've stumbled on another way to form expressions that span multiple lines.

Exercise 13: Python's exponentiation operator is **. For example, the expression that calculates 3^2 ("3, squared") is `3 ** 2`.

- Type the following expression at the shell prompt to calculate 3^2. Write the value Python displays when it evaluates the expression.

  ```
  >>> 3 ** 2
  ```

  _____

- Suppose we want to raise -3 to the power 2; i.e., calculate -3^2 ("-3, squared"), which equals 9.

  Perhaps the Python expression `-3 ** 2` would do this. Type this expression and record the value Python displays. Does this expression correctly calculate -3^2; that is, is the result 9?

  ```
  >>> -3 ** 2
  ```

  _____

- Based on what you observed, which operation does Python perform first when it evaluates the expression `-3 ** 2`: **exponentiation** (i.e., does Python first raise +3 to the power 2, yielding 9, then negate the result, yielding -9) or **negation** (i.e., does Python first negate +3, yielding -3, then raise -3 to the power 2, yielding 9)?

  _____

- Create some experiments to determine the simplest expression that raises -3 to the power 2, yielding 9; i.e., what is the simplest expression that squares -3? (For those of you who are reading ahead, don't use Python's `pow` function.) Record your experiments and your conclusions below (use another page if you need more room).

  _____

  _____

  _____

  _____

  _____

Exercise 14: What kind of numbers are 1, 2, 0, -1, etc.? Python has a built-in *function* called `type`, which, when *called*, provides information about the *type* of its *argument*. (For those who haven't programmed before, calling a function is analogous to entering a number on your handheld calculator (the argument), then pressing a function key labelled `type`). Type the following expressions and record the results:

```
>>> type(1)
```

_____

```
>>> type(2)
```

_____

```
>>> type(0)
```

_____

```
>>> type(-1)
```

_____

```
>>> type(255)
```

_____

```
>>> type(-256)
```

_____

```
>>> type(2 ** 31 - 1)
```

_____

```
>>> type((-2) ** 31)
```

_____

The last two expressions determine the type of 2^{31} - 1 (a very large integer) and -2^{31} (a very small integer.

The information displayed by Python after each of these function calls should be `<class 'int'>`. Did you observe anything different? If not, we can conclude that integers are values of type `int`. (Python types are implemented by a programming construct known as a *class*, so we can also say that Python integers are *instances* of class `int`.)

Exercise 15: Earlier, you learned that dividing two integers yields a real number result. What is the type of Python's real number values? Type these expressions and record the results:

```
>>> type(2.4)
```

_____

```
>>> type(0.0)
```

_____

```
>>> type(-11.2)
```

_____

The name of the type, `float`, is shorthand for *floating point*, which is one way that real numbers can be represented in a computer. You'll learn more about floating point representations in courses that deal with computer architecture, starting in Second Year.

Exercise 16: What operations are permitted with values of type `float`? Can Python add two real numbers? Subtract them? Multiply and divide them? If these operations are supported, what is the type of each result? For example, does `6.0 / 3.0` evaluate to 2 or 2.0? Type some expressions to determine if the calculations performed by the `+`, `-`, `*`, and `/` operators produce the results you'd expect, based on your knowledge of real number arithmetic from high school. On the ruled lines, record the expressions you type and the values that are displayed when the expressions are evaluated. (Use another page if you need more room.)

Exercise 17: Can we mix integers and real numbers in the same expression? For example, is `3.7 + 5` a valid Python expression? What about `5 + 3.7`? What about mixing integers and real numbers in expressions containing the subtraction, multiplication or division operators? Experiment! Record your experiments below. (Use another page if you need more room.)

Exercise 18: In mathematics, applying the *floor function* to a number x (denoted, $\lfloor x \rfloor$) yields the largest integer less than or equal to x. For example, the floor of 3.25 is 3, the floor of -3.25 is -4, and the floor of 3.0 is 3. For more information, read:
http://mathworld.wolfram.com/FloorFunction.html

Integer division is division in which the fractional part (remainder) is discarded and is sometimes denoted by the symbol, \. Integer division of integers a and b, $a \setminus b$, can be defined as $\lfloor a / b \rfloor$,where / denotes "normal" division. For example, 7 \ 2 is $\lfloor 7 / 2 \rfloor$, which is $\lfloor 3.5 \rfloor$, which equals 3.

Python has a *floor division* operator, which is represented by `//`. When Python evaluates the expression `a // b`, where `a` and `b` are integers, `a` is divided by by `b`, then the floor function is applied to the quotient. In other words, when its arguments are values of type `int`, the `//` operator performs integer division. For example,

- `15 // 2` yields 7 (15 / 2 yields 7.5, and $\lfloor 7.5 \rfloor$ is 7).
- `-15 // 2` yields -8 (-15 / 2 yields -7.5, and $\lfloor -7.5 \rfloor$ is -8).

When the operands are integers, the `//` operator always yields the largest integer that is less than or equal to the quotient; that is, it always rounds the quotient towards minus infinity.

Without using Python, predict the values of these expressions. Record your predictions.

```
11 // 4
```

_____

```
-11 // 4
```

_____

```
11 // -4
```

_____

```
-11 // -4
```

_____

Now use the Python shell to evaluate these expressions. Were your predictions correct?

Exercise 19: Recall from elementary school that the result of dividing two integers can be expressed as an integer quotient and an integer remainder. For example, 14 divided by 5 is 2 (the quotient), with a remainder of 4.

We've learned that Python's floor division operator calculates integer quotients when the operands are integers. For example typing:

```
>>> 14 // 5
```

yields 2, because 14 divided by 5 is 2.8, and the floor function applied to 2.8 yields 2.

Python's *modulo* operator, `%`, calculates the remainder that is "left over" when one integer is divided by another. At the shell prompt, type:

```
>>> 14 % 5
```

The result is 4 (because 14 divided by 5 yields remainder 4).

For any integers *x* and `y`, this identity is always true:

```
x == (x // y) * y + (x % y)
```

(Aside: `==` is the Python operator that compares its operands for equality. You can think of it as being equivalent to the mathematical equals sign, =.)

This identity tells us that Python's floor division (`x // y`) and modulo (`x % y`) operations are connected this way: the dividend, *x*, is equal to the integer quotient (`x // y`) multiplied by the divisor, *y*, plus the integer remainder (`x % y`).

Let's verify this for the case where `x` is `14` and `y` is `5`. At the shell prompt, type:

```
>>> (14 // 5) * 5 + (14 % 5)
```

As expected, the result is 14. (If you're not sure why, evaluate this expression without using Python.) So, we've verified that this identity is true:

```
14 == (14 // 5) * 5 + (14 % 5)
```

Things get interesting when one or both operands are negative. At the shell prompt, type:

```
>>> -14 // 5
```

The result is -3 (because -14 divided by 5 is -2.8, which is then rounded towards minus infinity, yielding -3.) So, what is the remainder; i.e., what is `-14 % 5`?

We know that the identity `x == (x // y) * y + (x % y)` must be satisfied, so substitute

-14 and 5 for *x* and *y* and simplify:

```
-14 == (-14 // 5) * 5 + (-14 % 5)
    == -3 * 5 + (-14 % 5)
    == -15 + (-14 % 5)
```

To satisfy the identity, `-14 % 5` must evaluate to 1.

Let's verify this. At the shell prompt, type:

```
>>> (-14 // 5) * 5 + (-14 % 5)
```

The result is -14, so clearly, the identity is satisfied.

At the shell prompt, type:

```
>>> -14 % 5
```

The result is 1, as we predicted.

Let's repeat this experiment for (14, -5). Type these expressions and record the results:

```
>>> 14 // -5
```

_____

```
>>> 14 % -5
```

_____

```
>>> (14 // -5) * -5 + (14 % -5)
```

_____

Can you conclude that the identity `14 == (14 // -5) * -5 + (14 % -5)` is true?

_____

Let's repeat this experiment for (-14, -5). Type these expressions and record the results:

```
>>> -14 // -5
```

_____

```
>>> -14 % -5
```

_____

```
>>> (-14 // -5) * -5 + (-14 % -5)
```

_____

Can you conclude that the identity `-14 == (-14 // -5) * -5 + (-14 % -5)` is true?

_____

Make sure you understand that, when `x` and `y` are integers, the expression:

```
(x // y) * y + (x % y)
```

always evaluates to `x`. If this isn't clear, go back and repeat the exercise to this point.

Without using Python, predict the values of these expressions. Record your predictions.

```
>>> (22 // 7) * 7 + (22 % 7)
```

_____

```
>>> 22 // 7
```

_____

```
>>> (22 // 7) * 7
```

_____

```
>>> 22 % 7
```

_____

This exercise continues on the next page.

```
>>> (-22 // 7) * 7 + (-22 % 7)
```

_____

```
>>> -22 // 7
```

_____

```
>>> (-22 // 7) * 7
```

_____

```
>>> -22 % 7
```

_____

```
>>> (22 // -7) * -7 + (22 % -7)
```

_____

```
>>> 22 // -7
```

_____

```
>>> (22 // -7) * -7
```

_____

```
>>> 22 % -7
```

_____

This exercise continues on the next page.

```
>>> (-22 // -7) * -7 + (-22 % -7)
```

_____

```
>>> -22 // -7
```

_____

```
>>> (-22 // -7) * -7
```

_____

```
>>> -22 % -7
```

_____

Now use the Python shell to determine if your predictions were correct.

Summary: given two integers *j* and *k*, write the steps you should follow (without using the Python interpreter) to predict the value Python will calculate when it evaluates the expression:

```
j % k
```

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____