UNIVERSITY OF WATERLOO

Lesson 1.2: Arithmetic in Python
Arithmetic in Python

In this lesson, we will discuss how to perform basic mathematical operations using arithmetic. As was hinted at in the previous lesson, Python's syntax for arithmetic rules follows the more common conventional infix notation, that is `1 + 2` is written instead of `(+ 1 2)`. We saw this when we discussed concatenation of strings which we also used the addition symbol to denote.

In this sense, addition is **overloaded**, that is, there are multiple different contracts that will work for this symbol. In fact, addition, subtraction, and multiplication work for any combination of `Int` and `Float`.

Whenever doing operations with `Float` type values, the result is always a `Float`.

Further, addition is also valid between two `Str` types. Lastly, multiplication of a `Str` and an `Int` type is also permitted in Python. What does this operation return?

Brackets work just like they do in ordinary arithmetic. Division and exponentiation are a bit more involved and we will discuss these later.

Try the following out in the visualizer. The visualizer is a tool to help you while you're learning about Python see how Python stores data on the backend. Simply click on the *Visualize* button below to see. What are the values of all the variables? What are the data types of all the variables? Notice that Python will first simplify the expression before assigning the final value to a variable.

```
x = 2 * (4 + 12)
y = x + 8
z = y * y
q = 2.5 + 4.2
w = "hi"
u = w + w
v = 3 * w
```

Visualize   Reset Code

Concept Check 1.2.1

1 point possible (graded)
Recall from the style guide the notation we used in CS 115 about contracts. Which of the following are valid contracts for the `+` operation? **Check all that apply.**

- ☐ `+: Int Int -> Int`
- ☐ `+: Int Float -> Int`
- ☐ `+: Float Float -> Float`

☐ `+: Float Int -> Float`

☐ `+: Str Str -> Str`

Concept Check 1.2.2

1 point possible (graded)
Consider the following code:

```
s = "hi"
t = 2 * s
```

What is the value of `t` ?

Division

Division is a bit more complicated in Python. The difficulty here is that the division of two integers is not always an integer. This is different from addition, multiplication, and subtraction where these operations, when given two integers, always return an integer. There is another issue of dividing by 0 that needs to be addressed as well.

To handle this issue, Python has two division operators:

1. Floating point division denoted by `/`.
2. Integer division denoted by `//`. This is equivalent to Racket's `quotient` function.

For floating point division `x/y` (also just known as division), if the types of `x` and `y` are either `Int` or `Float` values and `y` is not 0, then the result is the floating point number that results from the division. That is, the type of `x/y` is always `Float`. If `y` is 0, an error is produced and displayed to the screen. Try this out below!

Division By Zero

```
a = 1/0
```

Concept Check 1.2.3

1 point possible (graded)
What is the simplified value of `1/2` ?

◯ 0

◯ 0.0

◯ 0.5

◯ 1

◯ 1.0

Concept Check 1.2.4

1 point possible (graded)
What is the simplified value of `2/1` ?

◯ 1

◯ 1.0

◯ 2

◯ 2.0

Integer Division

For integer division `x//y` (also just known as division), if the types of `x` and `y` are `Int` and `y` is not 0, then the result is the `Int` that results from the usual mathematical division (rounded down if necessary). If however, one of `x` or `y` is a `Float`, then the type of `x//y` is a `Float`, assuming `y` is still non-zero. If `y` is 0, an error is produced and displayed to the screen.

For the purposes of this course, we **recommend only using integer division with integers**. The result of `x//y` is then the integer corresponding to the floating point value of `x/y` rounded down. So for example, `-11//4` and `9//-4` both give the value `-3` whereas the values of `11//4` and `-9//-4` both give 2. Notice that unary negation takes precedence before mathematical operators.

As a consequence of the above, when `x` and `y` are natural numbers and `y` is not 0, then the result is the quotient from the division algorithm.

Concept Check 1.2.5

1 point possible (graded)
Consider the code below

```
a = 17//5
b = -17//3
```

What are the values of `a` and `b` above?

◯ `a` is 3 and `b` is -6.

◯ `a` is 3 and `b` is -5.

◯ `a` is 4 and `b` is -6.

◯ `a` is 4 and `b` is -5.

◯ None of the above

Remainder

In this course, we will only use `x % y` when `x` is a `Nat` and `y` is a positive integer. In this case, the `%` operator (pronounced modulo operator) is precisely the remainder when we divide `x` by `y` using the Division Algorithm. For example, `5 % 3` is `2` since $5 = 1(3) + 2$, that is, three divides into 5 once with a remainder of 2.

In terms of order of operations, the `%` is at the same precedence level as multiplication and division. The `%` operation is equivalent to Racket's `remainder` function.

Concept Check 1.2.6

1 point possible (graded)
What is the value of `9973 % 10`?

Exponents

I always like to quip that exponentiation is twice as strong as multiplcation and so the symbol we use for exponentiation in Python is two * symbols given by ** (many other programming languages use ^ instead!) Exponentiation in Python can be done with either integers or floating point numbers and the result is an integer if two integers are given as parameters and otherwise is of type `Float`.

With exponentiation complete, we can now fully recap the order of operations (BEDMAS) with respect to the above operations:

- Brackets (or parentheses) are evaluated first
- Exponents are evaluated next
- Multiplication, Division, and Modulo are evaluated next
- Addition and Subtraction are evaluated last

For those wondering, unary negation occurs just after evaluating exponents. If there is a tie with operations, they are evaluated in the order of the associativity of the operator (more on this at the end of this lesson).

Concept Check 1.2.7

1 point possible (graded)
What is the value of `2 * 1 ** 2`?

Concept Check 1.2.8

3 points possible (graded)
Consider the following code:

```
w = 5-4*3
```

```
y = 28//10
z = 96 % 10
```

What is the value of `w` ?

What is the value of `y` ?

What is the value of `z` ?

Concept Check 1.2.9

1 point possible (graded)
What is the type and value of `22/2` ?

- ○ `Int` and `11`
- ○ `Int` and `11.0`
- ○ `Float` and `11`
- ○ `Float` and `11.0`

Associativity

Associativity refers to the order in which we do operations. In particular, we focus on associativity with arithmetic rules here. With arithmetic rules that are **left associative**, operations are executed left to right. With arithmetic rules that are **right associative**, operations are executed right to left. This might seem confusing at first but the following two examples might help to illuminate the situation. Note that most, but not all rules are left associative as we will see below.

Concept Check 1.2.10

1 point possible (graded)
Consider the following code:

```
a = 1 - 2 - 3
```

What is the value of `a` ?

Concept Check 1.2.11

1 point possible (graded)
Consider the following code:

```
b = 2 ** 1 ** 2
```

What is the value of `b`?

Notice how above, the first example was evaluated using left associativity. That is, a gets the value of $(1-2)-3$ instead of $1-(2-3)$. However, exponentiation is right associative! Meaning `b` got the value $2**(1**2)$ instead of $(2**1)**2$. At first, this might seem weird but notice when we write exponent for an expression like 2^{1^2}, we actually want to evaluate the 1^2 value first and then evaluate 2^1. So this actually matches how we do arithmetic very well.

You will not be tested on associativity in this course and is mentioned only if you happen to come across it in your own code. This might be confusing but there is always a simple solution to fix this: use brackets to force a specific ordering of operations you want to execute!

Even with associativity however, arithmetic expressions must first obey the order of operations so an expression like `2*3 % 5*2` is evaluated as`((2*3) % 5)*2` independent of the white spaces put in the expression instead of like `(2*3) % (5*2)`. Again remember if there is any ambiguity, use brackets!

Concept Check 1.2.12

1 point possible (graded)
What is the value of `2*3 % 5*2`?

A Final Warning About Floats

`Float` values are inexact. This means the final answers are often approximations to the mathematical equivalent final answer. Python has difficulties storing values like `1/3` since it has to truncate the decimal value but perhaps less obvious at this point is that Python also struggles storing larger values as well. While Python can store integers exactly and arbitrarily large, it cannot do the same with floating point numbers due to how the internal representation of these numbers are stored. You don't need to know the exact details of this but what is important in this course is whenever you have a choice between doing arithmetic with integers and with floating point numbers **always do the arithmetic over the integers to avoid rounding errors**.

Whenever doing operations with `Float` type values, expect that the final answer is at best an approximation to the mathematical answer. This is true for computations with extremely small values **and** extremely large values.

Final Thoughts

In this lesson, we introduced all the basic built-in arithmetic operations. In the next lesson, we will discuss how to use and create more complicated functions. We then will do a full review of the design recipe and discuss what modifications we need to make in order to make our design recipe from our previous course work in this course. Lastly, we will complete this module by introducing the `math` module, which has even more arithmetic operations!