Expressions and Data Types

Now that you have looked at arithmetic operators, you will see how operators and operands can be combined to form expressions. In particular, you will learn how arithmetic expressions are evaluated in Python.

Part I - Expressions

An **expression** is a combination of symbols that evaluates to a value. Expressions, most commonly, consist of a combination of operators and operands,

```
4 + (3 * k)
```

An expression can also consist of a single literal or variable. Thus, `4`, `3`, and `k` are each expressions. This expression has two *subexpressions*, `4` and `(3 * k)`. Subexpression `(3 * k)` itself has two subexpressions, `3` and `k`.

Expressions that evaluate to a numeric type are called **arithmetic expressions**. A **subexpression** is any expression that is part of a larger expression. Subexpressions may be denoted by the use of parentheses, as shown above. Thus, for the expression `4 + (3 * 2)`, the two operands of the addition operator are `4` and `(3 * 2)`, and thus the result is equal to `10`. If the expression were instead written as `(4 + 3) * 2`, then it would evaluate to `14`.

Since a subexpression is an expression, any subexpression may contain subexpressions of its own,

```
4 + (3 * (2 - 1)) → 4 + (3 * 1) → 4 + 3 → 7
```

If no parentheses are used, then an expression is evaluated according to the rules of operator precedence in Python, discussed in the next section.

Your Turn

From the Python Shell, enter the following and observe the results.

```
>>> (2 + 3) * 4
???

>>> 2 + ((3 * 4 - 8)
???

>>> 2 + (3 * 4)
???

>>> 2 + 3 * (4 - 1)
???
```

Part II - Operator Precedence

The way we commonly represent expressions, in which operators appear between their operands, is referred to as **infix notation**. For example, the expression `4 + 3` is in infix notation since the `+` operator appears between its two operands, `4` and `3`. There are other ways of representing expressions called *prefix* and *postfix* notation, in which operators are placed *before* and *after* their operands, respectively.

The expression `4 + (3 * 5)` is also in infix notation. It contains two operators, `+` and `*`. The parentheses denote that `(3 * 5)` is a subexpression. Therefore, `4` and `(3 * 5)` are the operands of the addition operator, and thus the overall expression evaluates to 19. What if the parentheses were omitted, as given below?

```
4 + 3 * 5
```

How would this be evaluated? These are two possibilities,

```
4 + 3 * 5   →   4 + 15   → 19
4 + 3 * 5   →   7 * 5 →  35
```

Some might say that the first version is the correct one by the conventions of mathematics. However, each programming language has its own rules for the order that operators are applied, called **operator precedence**, defined in an **operator precedence table**. This may or may not be the same as in mathematics, although it typically is. In the table below, the Python operator precedences discussed so far are defined.

Operator	Associativity
** (exponentiation)	right-to-left
- (negation)	left-to-right
* (mult), / (div), // (truncating div), % (modulo)	left-to-right
+ (addition), - (subtraction)	left-to-right

In the table, higher-priority operators are placed above lower-priority ones. Thus, we see that multiplication is performed before addition when no parentheses are included,

```
4 + 3 * 5 → 4 + 15 →  19
```

In our example, therefore, if the addition is to be performed first, parentheses would be needed,

```
(4 + 3) * 5 → 7 * 5 →      35
```

As another example, consider the expression below. Following Python's rules of operator precedence, the exponentiation operator is applied first, then the truncating division operator, and finally the addition operator,

```
4 + 2 ** 5 // 10 → 4 + 32 // 10 → 4 + 3 →  7
```

Operator precedence guarantees a consistent interpretation of expressions. However, it is good programming practice to use parentheses even when not needed if it adds clarity and enhances readability, without overdoing it. Thus, the previous expression would be better written as,

```
4 + (2 ** 5) // 10
```

Your Turn

From the Python Shell, enter the following and observe the results.

```
>>> 2 + 3 * 4
???

>>> 2 * 3 // 4
???

>>>2 * 3 + 4
???

>>> 5 + 42 % 10
???

>>> 2 * 3 / 4
???

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

Part III - Operator Associativity

A question that you may have already had is, “What if two operators have the same level of precedence, which one is applied first?” For operators following the associative law, the order of evaluation doesn’t matter,

```
(2 + 3) + 4 → 9          2 + (3 + 4) → 9
```

In this case, we get the same results regardless of the order that the operators are applied. Division and subtraction, however, do not follow the associative law,

a) `(8 - 4) - 2 → 4 - 2 → 2` `8 - (4 - 2) → 8 - 2 → 6`
b) `(8 / 4) / 2 → 2 / 2 → 1` `8 / (4 / 2) → 8 / 2 → 4`
c) `2 ** (3 ** 2) → 512` `(2 ** 3) ** 2 → 64`

Here, the order of evaluation does matter. To resolve the ambiguity, each operator has a specified **operator associativity** that defines the order that it and other operators with the same

level of precedence are applied (as given in the previous table). All operators in the figure, except for exponentiation, have left-to-right associativity—exponentiation has right-to-left associativity.

Your Turn

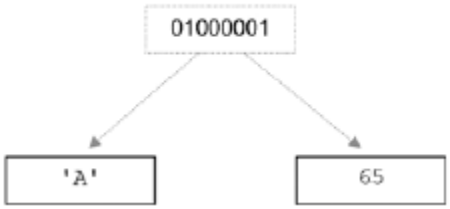
From the Python Shell, enter the following and observe the results.

```
>>>6 - 3 + 2          >>>2 * 3 / 4          >>> (2 ** 2) ** 3
???                   ???                   ???

>>>(6 - 3 + 2         >>> 12 % (10 / 2)     >>> 2 ** (2 ** 3)
 ???                  ???                   ???

>>> 6 - (3 + 2)       >>> 2 ** 2 ** 3
???                   ???
```

Part IV - Expressions

A **data type** is a set of values, and a set of operators that may be applied to those values. For example, the integer data type consists of the set of integers, and operators for addition, subtraction, multiplication, and division, among others. Integers, floats, and strings are part of a set of predefined data types in Python called the **built-in types**.

Data types prevent the programmer from using values inappropriately. For example, it does not make sense to try to divide a string by two, `'Hello' / 2`. The programmer knows this by common sense. Python knows it because `'Hello'` belongs to the string data type, which does not include the division operation. The need for data types results from the fact that the same internal representation of data can be interpreted in various ways, as shown below:



The sequence of bits in the figure can be interpreted as a character (`'A'`) or an integer (`65`). If a programming language did not keep track of the intended type of each value, then the programmer would have to. This would likely lead to undetected programming errors, and would provide even more work for the programmer. We discuss this further in the following section.

Finally, there are two approaches to data typing in programming languages. In **static typing**, a variable is declared as a certain type before it is used, and can only be assigned values of that type. Python, however, uses *dynamic typing*. In **dynamic typing**, the data type of a variable depends only on the type of value that the variable is currently holding. Thus, the same variable may be assigned values of different type during the execution of a program.

Mixed-Type Expressions

A **mixed-type expression** is an expression containing operands of different type. The CPU can only perform operations on values with the same internal representation scheme, and thus only on operands of the same type. Operands of mixed-type expressions therefore must be converted to a com- mon type. Values can be converted in one of two ways—by implicit (automatic) conversion, called *coercion*, or by explicit *type conversion*. We look at each of these next.

Coercion vs. Type Conversion

Coercion is the *implicit* (automatic) conversion of operands to a common type. Coercion is automatically performed on mixed-type expressions only if the operands can be safely converted, that is, if no loss of information will result. The conversion of integer `2` to floating-point `2.0` below is a safe conversion—the conversion of `4.5` to integer `4` is not, since the decimal digit would be lost,

`2` + `4.5` → **`2.0`** + `4.5` → `6.5` safe (automatic conversion of `int` to `float`)

Type conversion is the *explicit* conversion of operands to a specific type. Type conversion can be applied even if loss of information results. Python provides built-in **type conversion functions** `int()` and `float()`, with the `int()` function truncating results as given in Figure 2-21.

```
float(2) + 4.5  →  2.0 + 4.5  →  6.5
2 + int(4.5) →  2 + 4 →     6
```

Conversion Function		Converted Result		Conversion Function		Converted Result
`int()`	`int(10.8)`	10		`float()`	`float(10)`	10.0
	`int('10')`	10			`float('10')`	10.0
	`int('10.8')`	ERROR			`float('10.8')`	10.8

Note that numeric strings can also be converted to a numeric type. In fact, we have already been doing this when using `int` or `float` with the `input` function,

```
num_credits = int(input('How many credits do you have? '))
```

Concepts and Routines

1. What value does the following expression evaluate to?

```
2 + 9 * ((3 * 12) – 8) / 10
```

(a) 27 **(b)** 27.2 **(c)** 30.8

2. Evaluate the following arithmetic expressions using the rules of operator precedence in Python.

a) `3 + 2 * 10`

b) `2 + 5 * 4 + 3`

c) `20 // 2 * 5`

d) `2 * 3 ** 2`

3. Evaluate the following arithmetic expressions based on Python's rules of operator associativity.

a) `24 // 4 // 2`

b) `2 ** 2 ** 3`

4. Which of the following is a mixed-type expression?

a) `2 + 3.0`

b) `2 + 3 * 4`

5. Which of the following would involve coercion when evaluated in Python?

a) `4.0 1 3`

b) `3.2 * 4.0`

6. Which of the following expressions use explicit type conversion?

a) `4.0 1 float(3)`

b) `3.2 * 4.0`

c) `3.2 + int(4.0)`

Problem Solving

1. Evaluate the following expressions in Python.

a) `10 - (5 * 4)`

b) `40 % 6`

c) `2(10 / 3) + 2`

2. Give all the possible evaluated results for the following arithmetic expression (assuming no rules of operator precedence).

`2 * 4 + 25 – 5`

3. Parenthesize all of the subexpressions in the following expressions following operator precedence in Python.

a) `var1 * 8 2var2 + 32 / var3`

b) `var1 -6 ** 4 * var2 ** 3`

4. Evaluate each of the expressions in question 17 above for `var1 = 10`, `var2 = 30`, and `var3 = 2`.

5. For each of the following expressions, indicate where operator associativity of Python is used to resolve ambiguity in the evaluation of each expression.

a) `var1 * var2 * var3 -var4`

b) `var1 * var2 / var3`

c) `var1 ** var2 ** var3`

6. Using the built-in type conversion function `float()`, alter the following arithmetic expressions so that each is evaluated using floating-point accuracy. Assume that `var1`, `var2`, and `var3` are assigned integer values. Use the minimum number of calls to function `float()` needed to produce the results.

a) `var1 1 var2 * var3`

b) `var1 // var2 1var3`

c) `var1 // var2 / var3`