PES UNIVERSITY

PES UNIVERSITY

UE18CS101: INTRODUCTION TO COMPUTING USING python™

Department of Computer Science and Engineering
PES UNIVERSITY

Lecture 8 and 9

Operators and Expressions

Precedence and Associativity

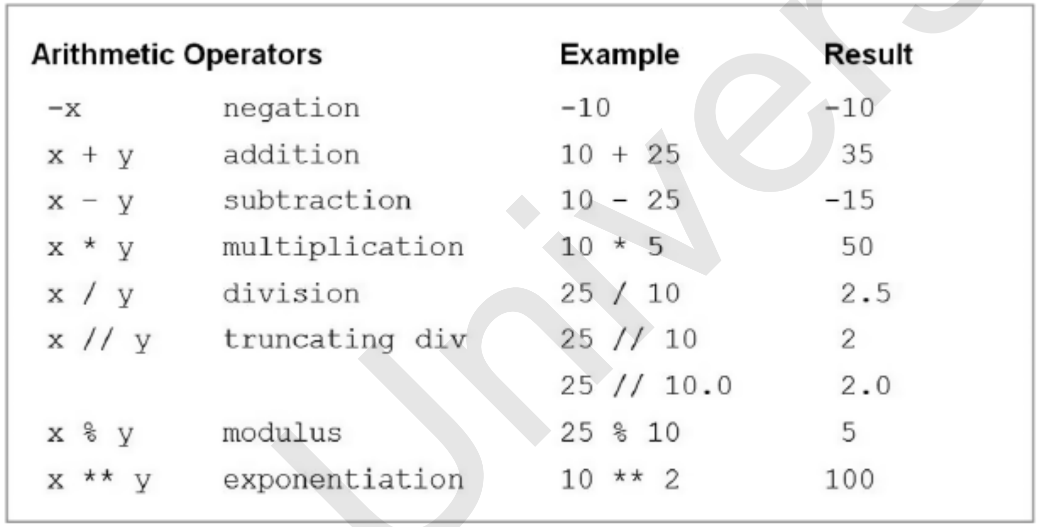
Operators

An operator is a symbol that represents an operation that may be performed on one or more *operands*. For example, the **+** symbol represents the operation of addition. **An operand is a value that a given operator is applied to,** such as operands 2 and 3 in the expression 2 + 3.

A **unary operator** operates on only one operand, such as the negation operator in the expression - 12.

A binary operator operates on two operands, as with the addition operator. Most operators in programming languages are binary operators. We look at the arithmetic operators in Python next.

Arithmetic Operators



Arithmetic Operators		Example	Result
`-x`	negation	`-10`	`-10`
`x + y`	addition	`10 + 25`	`35`
`x - y`	subtraction	`10 - 25`	`-15`
`x * y`	multiplication	`10 * 5`	`50`
`x / y`	division	`25 / 10`	`2.5`
`x // y`	truncating div	`25 // 10`	`2`
		`25 // 10.0`	`2.0`
`x % y`	modulus	`25 % 10`	`5`
`x ** y`	exponentiation	`10 ** 2`	`100`

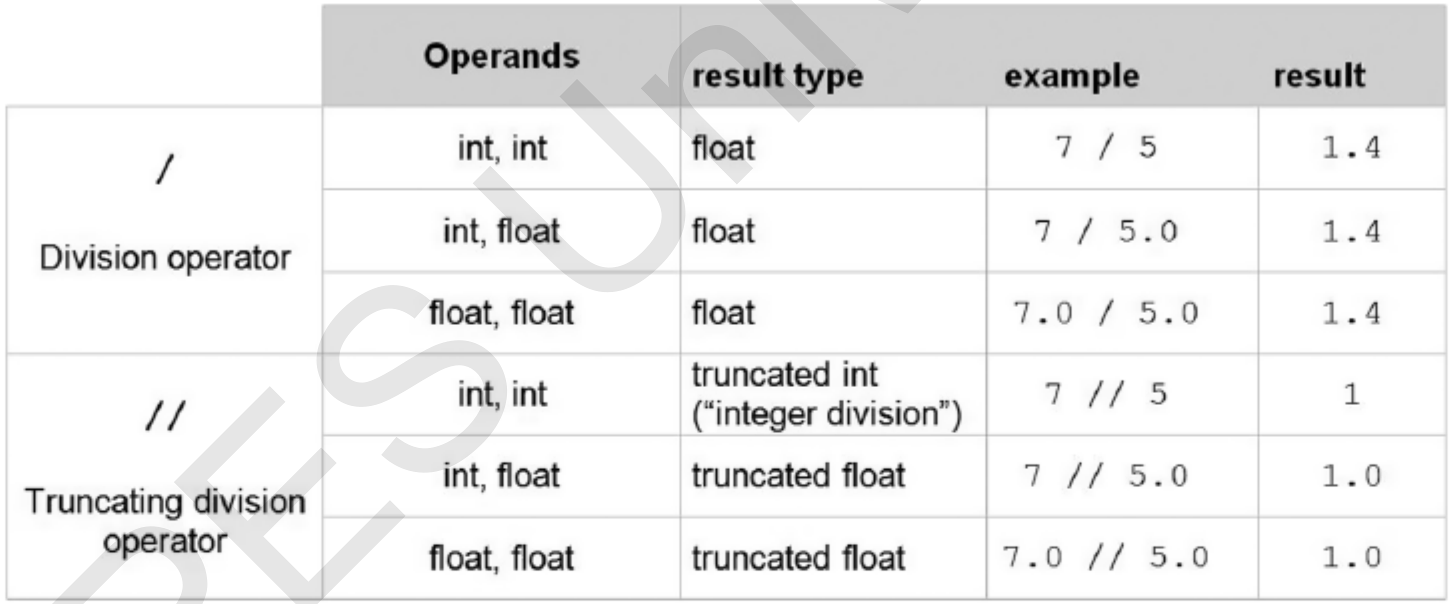
The **+** , **-**, ***** (**multiplication**) and **/** (**division**) arithmetic operators perform the usual operations. Note that the **–** symbol is used both as a unary operator (for negation) and a binary operator (for subtraction),

```
20 - 5 → 15          (– as binary operator)
- 10 * 2 → - 20      (– as unary operator)
```

Python also includes an **exponentiation** (******) operator.

Python provides two forms of division. **"True" division** is denoted by a single slash, **/**. Thus, **25 / 10 evaluates to 2.5**. **Truncating division** is denoted by a double slash, **//**, providing a truncated result based on the type of operands applied to.

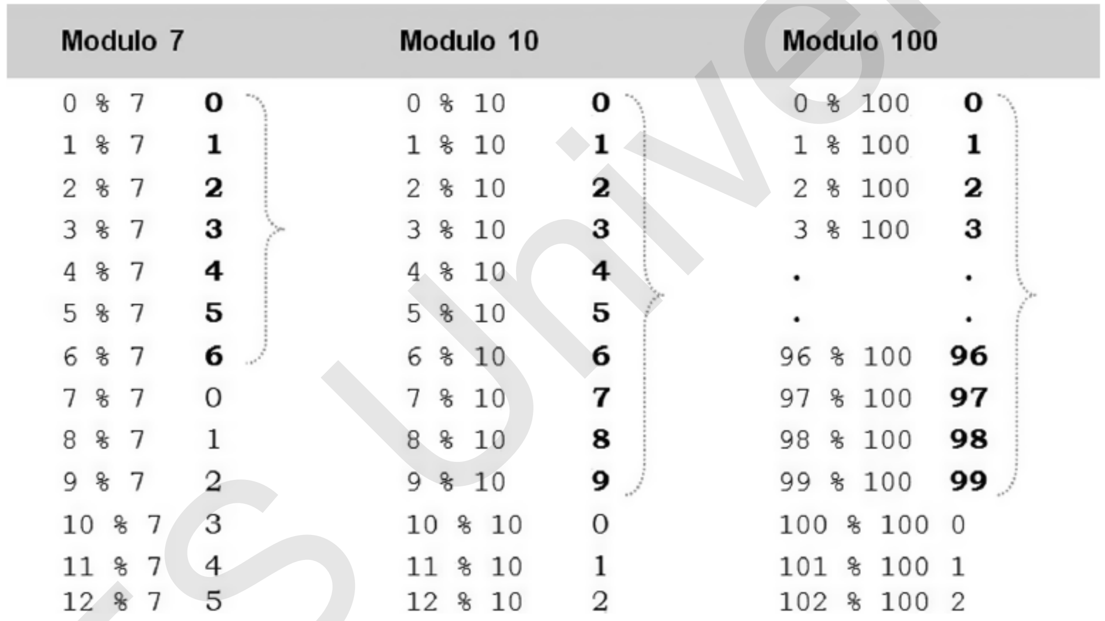
When both operands are integer values, the result is a truncated integer referred to as **integer division**. When as least one of the operands is a float type, the result is a **truncated floating point**. Thus, **25 // 10 evaluates to 2**, while **25.0 // 10 evaluates to 2.0**.



	Operands	result type	example	result
/ Division operator	int, int	float	`7 / 5`	`1.4`
	int, float	float	`7 / 5.0`	`1.4`
	float, float	float	`7.0 / 5.0`	`1.4`
// Truncating division operator	int, int	truncated int ("integer division")	`7 // 5`	`1`
	int, float	truncated float	`7 // 5.0`	`1.0`
	float, float	truncated float	`7.0 // 5.0`	`1.0`

As example use of integer division, the number of dozen doughnuts for variable `numDoughnuts` = 29 is: `numDoughnuts // 12` → `29 // 12` → `2`

Lastly, the modulus operator (`%`) **gives the remainder of the division of its operands**, resulting in a cycle of values.



Modulo 7		Modulo 10		Modulo 100	
0 % 7	**0**	0 % 10	**0**	0 % 100	**0**
1 % 7	**1**	1 % 10	**1**	1 % 100	**1**
2 % 7	**2**	2 % 10	**2**	2 % 100	**2**
3 % 7	**3**	3 % 10	**3**	3 % 100	**3**
4 % 7	**4**	4 % 10	**4**	.	.
5 % 7	**5**	5 % 10	**5**	.	.
6 % 7	**6**	6 % 10	**6**	96 % 100	**96**
7 % 7	0	7 % 10	**7**	97 % 100	**97**
8 % 7	1	8 % 10	**8**	98 % 100	**98**
9 % 7	2	9 % 10	**9**	99 % 100	**99**
10 % 7	3	10 % 10	0	100 % 100	0
11 % 7	4	11 % 10	1	101 % 100	1
12 % 7	5	12 % 10	2	102 % 100	2

The modulus and truncating (integer) division operators are complements of each other. For example, `29 // 12` **gives the number of dozen doughnuts**, while `29 % 12` **gives the number of leftover doughnuts** (**5**).

Expressions and Data Types

Now that we have looked at arithmetic operators, **we will see how operators and operands can be combined to form *expressions***. In particular, we will look at how arithmetic expressions are evaluated in Python. We also introduce the notion of a ***data type***.

What Is an Expression?

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.

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?

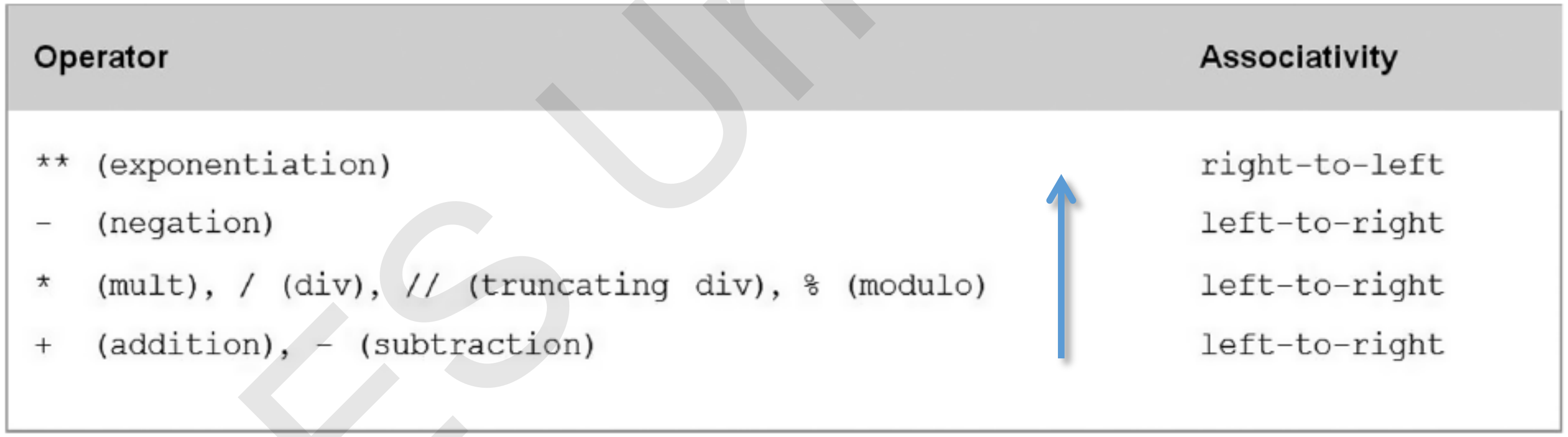
4 + 3 * 5

How would this be evaluated? These are two possibilities.

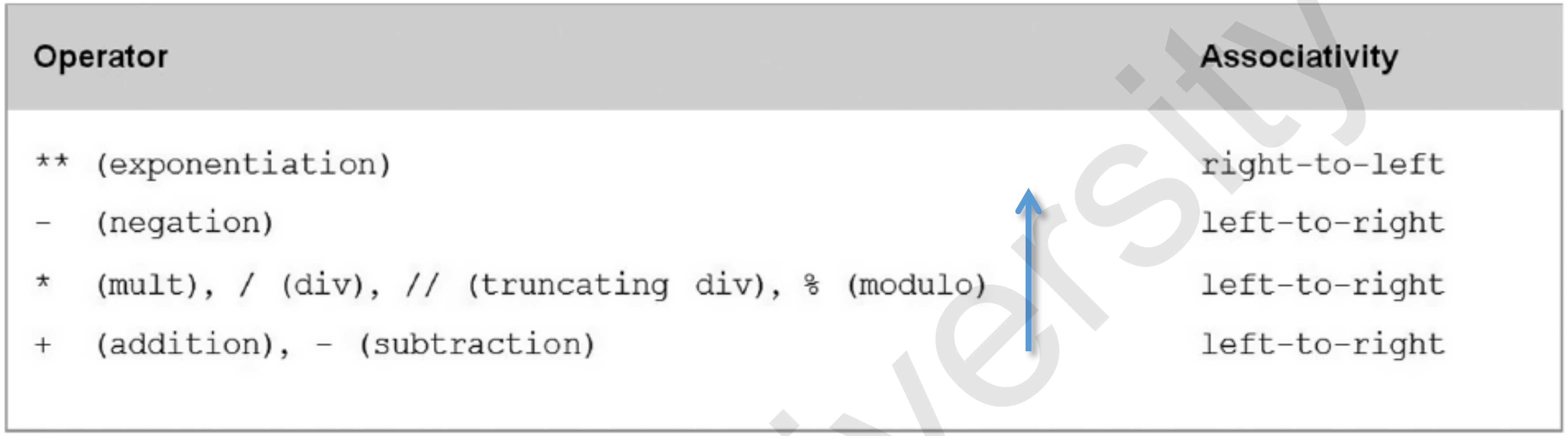
4 + 3 * 5 → 4 + 15 → 19 4 + 3 * 5 → 7 * 5 → 35

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.

Below is the operator precedence table for the Python operators discussed so far.



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



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,

$$4 + 3 * 5 \rightarrow 4 + 15 \rightarrow \mathbf{19}$$

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

$$(4 + 3) * 5 \rightarrow 7 * 5 \rightarrow \mathbf{35}$$

As another example, consider the expression below.

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

Following Python’s rules of operator precedence, the exponentiation operator is applied first, then the truncating division operator, and finally the addition operator.

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

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 (such as addition) 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,

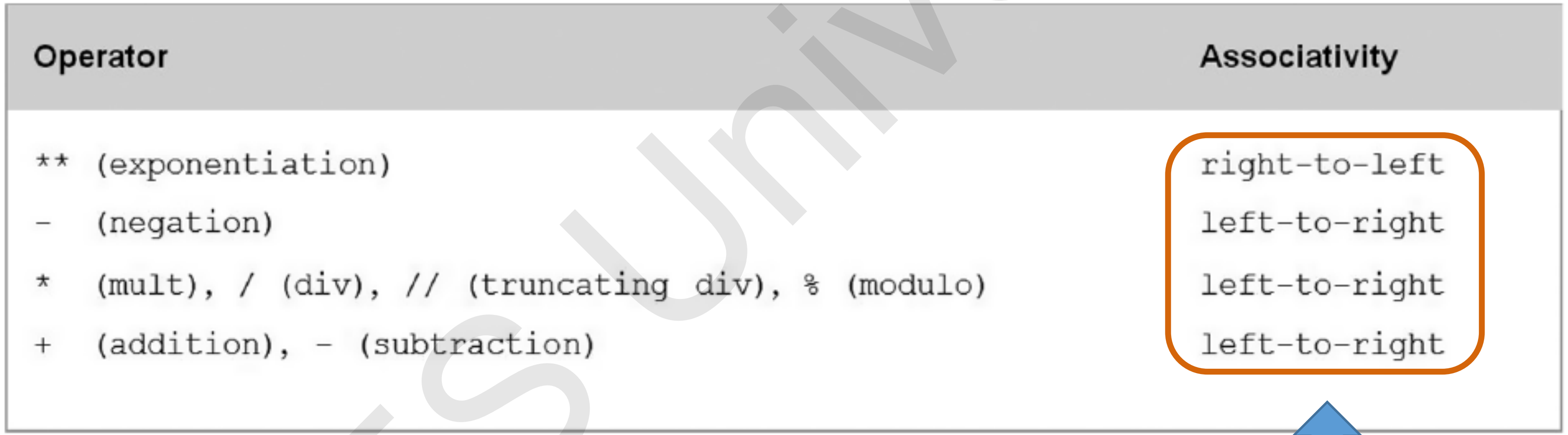
(a) (8 - 4) - 2 → 4 - 2 → **2**   8 - (4 - 2) → 8 - 2 → **6**

(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. **All operators given below, except for exponentiation, have left-to-right associativity—exponentiation has right-to-left associativity**.



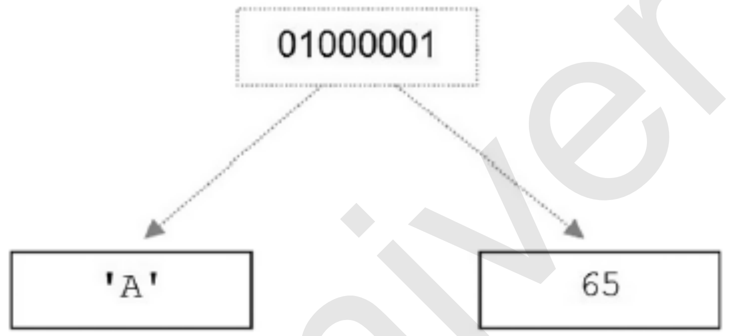
Operator	Associativity
`**` (exponentiation)	right-to-left
`-` (negation)	left-to-right
`*` (mult), `/` (div), `//` (truncating div), `%` (modulo)	left-to-right
`+` (addition), `-` (subtraction)	left-to-right

Data Types

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 .**

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,



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.

*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.

Python uses dynamic typing.

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 common type**. **Values can be converted in one of two ways**—by implicit (automatic) conversion, called ***coercion***, or by explicit ***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,

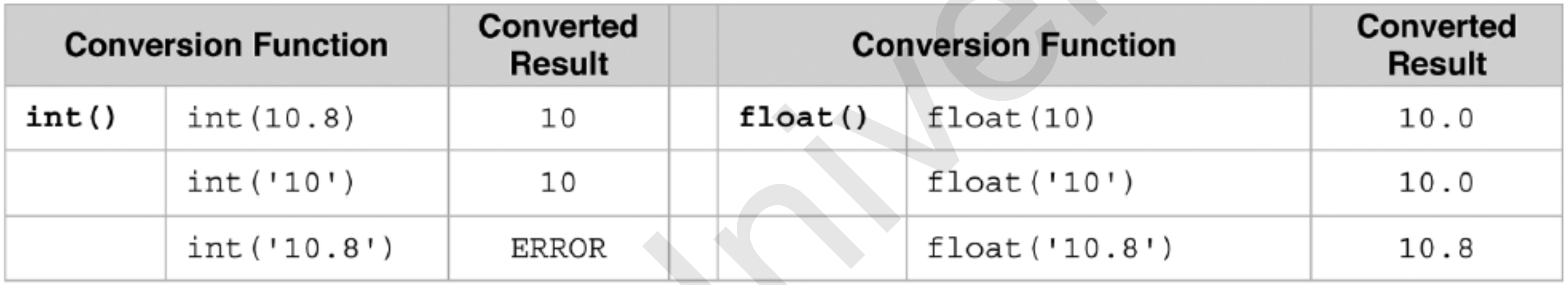
2	+	4.5	→	2.0	+	4.5	→	**6.5**	**safe**	(automatic conversion of int to float)
int		**float**		**float**		**float**		**float**		

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

2	+	4.5	→	float(2)	+	4.5	→	2.0	+	4.5	→	**6.5**	**No loss of information**
int		**float**		**float**		**float**		**float**		**float**		**float**	

2	+	4.5	→	2	+	int(4.5)	→	2	+	4	→	**6**	**Loss of information**
int		**float**		**int**		**int**		**int**		**int**		**int**	

Type conversion functions int() and float()



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? '))
```

Boolean Expressions

The **Boolean data type** contains two Boolean values, denoted as **True** and **False** in Python.

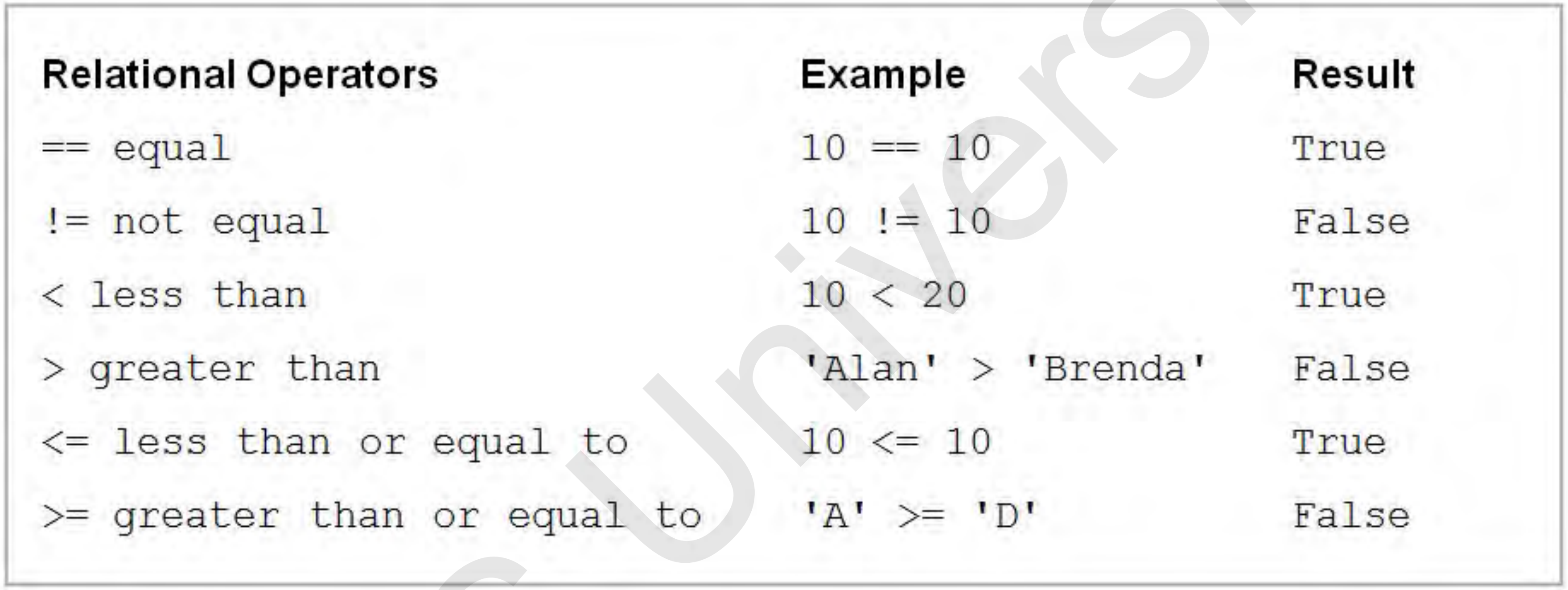
A **Boolean expression** is an expression that evaluates to a Boolean value. Boolean expressions are used to denote the conditions for selection and iterative control statements.

Relational Operators

The **relational operators** in Python perform the usual comparison operations.

Relational expressions are a type of **Boolean expression**, since they evaluate to a Boolean result.

Relational Operators in Python

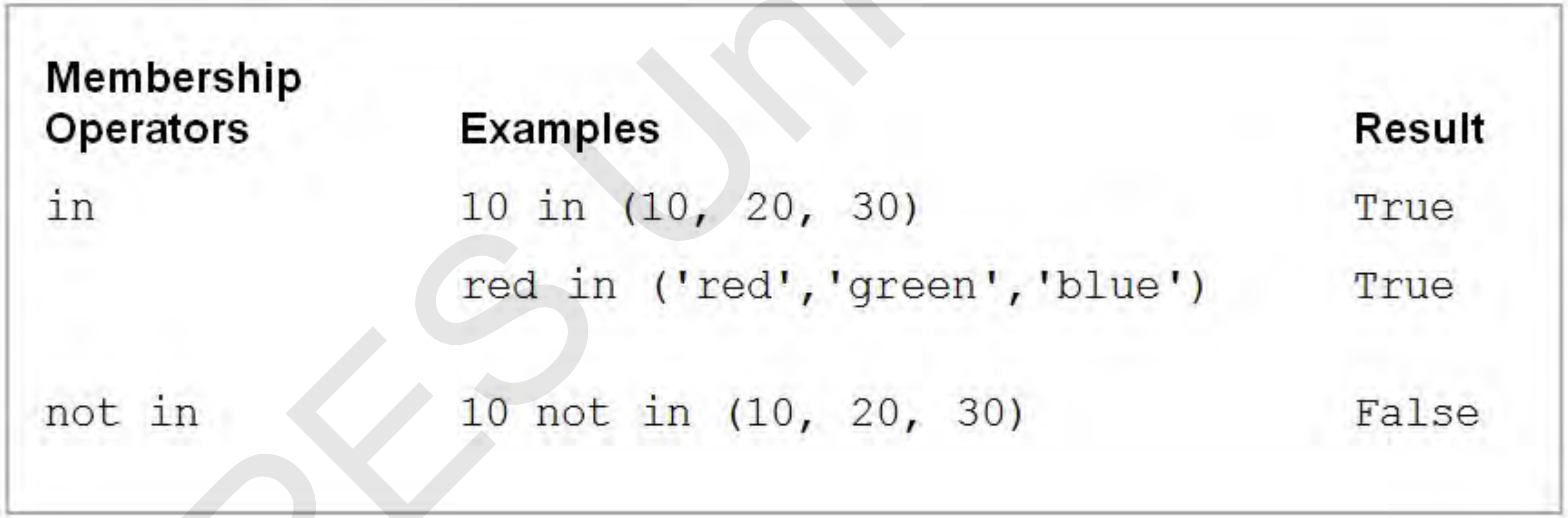


Relational Operators	Example	Result
`==` equal	`10 == 10`	True
`!=` not equal	`10 != 10`	False
`<` less than	`10 < 20`	True
`>` greater than	`'Alan' > 'Brenda'`	False
`<=` less than or equal to	`10 <= 10`	True
`>=` greater than or equal to	`'A' >= 'D'`	False

Note that these operators not only apply to numeric values, but to any set of values that has an ordering, such as strings.

Membership Operators

Python provides a convenient pair of **membership operators**. These operators can be used to easily determine if a particular value occurs within a specified list of values.



Membership Operators	Examples	Result
`in`	`10 in (10, 20, 30)`	`True`
	`red in ('red','green','blue')`	`True`
`not in`	`10 not in (10, 20, 30)`	`False`

The membership operators *can also be used to check if a given string occurs within another string,*

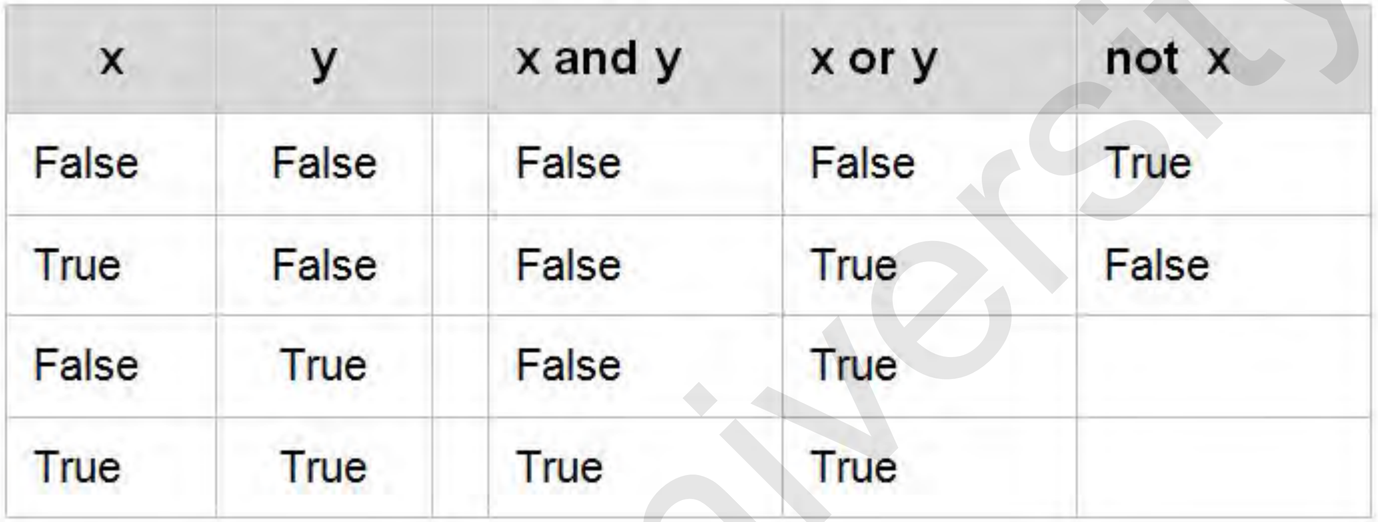
```
>>> 'Dr.' in 'Dr. Madison'
True
```

As with the relational operators, the membership operators can be used to construct Boolean expressions.

Boolean Operators

George Boole, in the mid-1800s, developed what we now call **Boolean algebra**. His goal was to develop an algebra based on true/false rather than numerical values.

Boolean algebra contains a set of **Boolean (logical) operators**, denoted by and, or, and not. These logical operators can be used to construct arbitrarily complex Boolean expressions.



x	y	x and y	x or y	not x
False	False	False	False	True
True	False	False	True	False
False	True	False	True	
True	True	True	True	

Logical and is true only when *both* its operands are true—otherwise, it is false. Logical or is true when *either or both* of its operands are true, and thus false only when both operands are false. Logical not simply reverses truth values—not False equals True, and not True equals False.

Consider the following :

1 <= num <= 10

Although in mathematics this notation is understood, consider how this would be evaluated in a programming language (for num equal to 15):

1 <= num <= 10 → 1 <= 15 <= 10 → True <= 10 → **?!?**

The subexpression for the left relational operator would be evaluated first, which evaluates to True. Continuing, however, it doesn't make sense to check if True is less than or equal to 10. Some programming languages would generate a mixed-type expression error for this.

Therefore, the correct way for computer evaluation of the condition is by use of the Boolean and operator (again for num equal to 15):

1 <= num and num <= 10 → (1 <= num) and (num <= 10) →

True and (num <= 10) → True and True → True

Let's see what we get when we do evaluate the expression in the Python shell (for num equal to 15)

```
>>> 1 <= num and num <= 10
False
```

We actually get the correct result, False. If we were to try the original form of the expression:

```
>>> 1 <= num  <= 10
False
```

This also works without error in Python?!

What you need to be aware of is that Python allows a relational expression of the form,

1 <= num <= 10

but and automatically converts it to the form before evaluated,

1 <= num and num <= 10

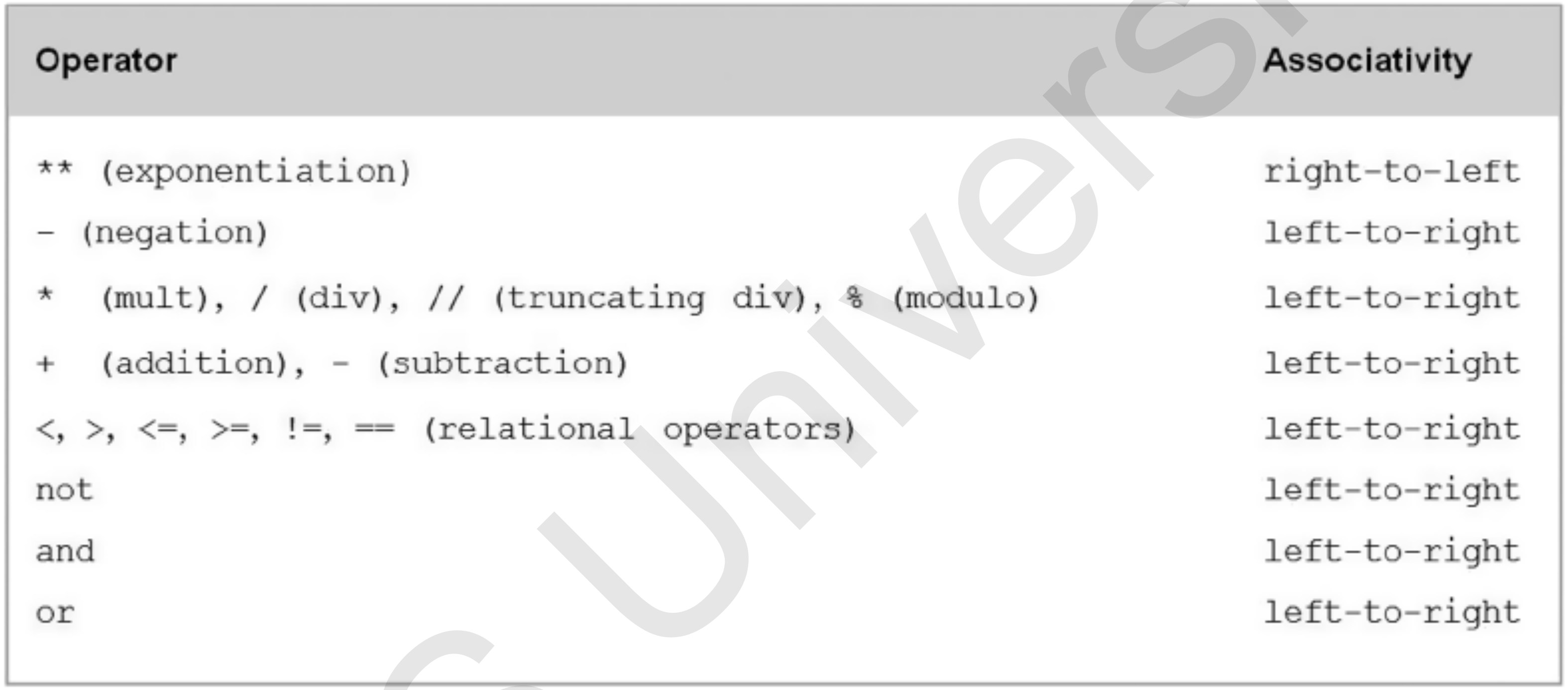
Thus, although Python offers this convenient shorthand, many programming languages require the longer form expression by use of logical and, and would give an error (or incorrect results) if written in the shorter form. Thus, as a novice programmer, *it would be best not to get in the habit of using the shorter form expression particular to Python*.

Operator Precedence and Boolean Expressions

George Boole, in the mid-1800s, developed what we now call **Boolean algebra**. His goal was to develop an algebra based on true/false rather than numerical values.

Boolean algebra contains a set of **Boolean (logical) operators**, denoted by and, or, and not. These logical operators can be used to construct arbitrarily complex Boolean expressions.

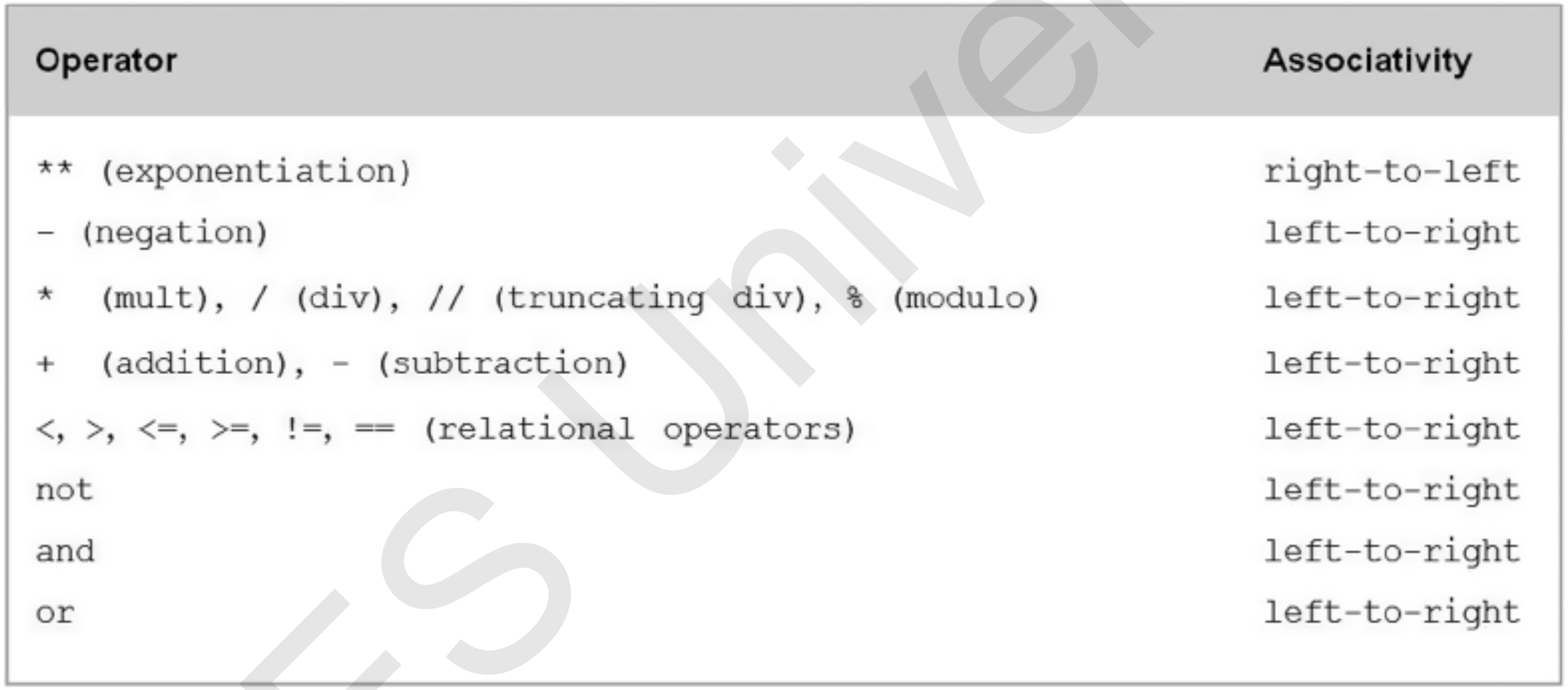
Operator Precedence and Boolean Expressions



Operator	Associativity
`**` (exponentiation)	right-to-left
`-` (negation)	left-to-right
`*` (mult), `/` (div), `//` (truncating div), `%` (modulo)	left-to-right
`+` (addition), `-` (subtraction)	left-to-right
`<`, `>`, `<=`, `>=`, `!=`, `==` (relational operators)	left-to-right
not	left-to-right
and	left-to-right
or	left-to-right

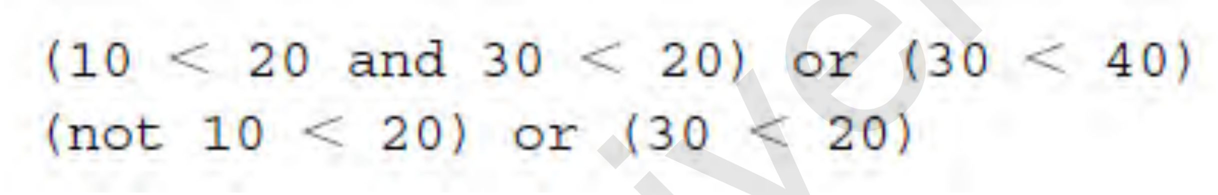
As we saw earlier, in the table, high-priority operators are placed before lower-priority operators. Thus we see that all arithmetic operators are performed before any relational or Boolean operators.

Unary Boolean operator not has higher precedence then and, and Boolean operator and has higher precedence than the or operator.



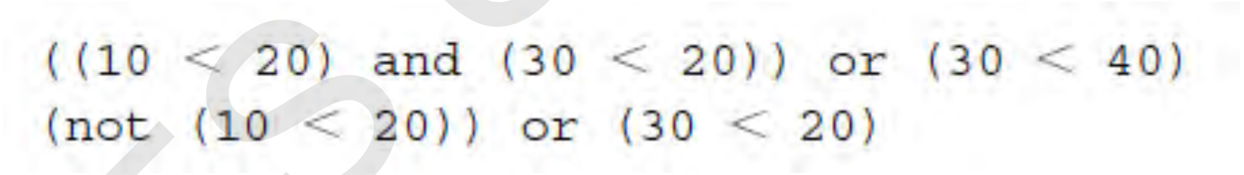
Operator	Associativity
`**` (exponentiation)	right-to-left
`-` (negation)	left-to-right
`*` (mult), `/` (div), `//` (truncating div), `%` (modulo)	left-to-right
`+` (addition), `-` (subtraction)	left-to-right
`<`, `>`, `<=`, `>=`, `!=`, `==` (relational operators)	left-to-right
not	left-to-right
and	left-to-right
or	left-to-right

As with arithmetic expressions, it is good programming practice to use parentheses, even if not needed, to add clarity and enhance readability,



```
(10 < 20 and 30 < 20) or (30 < 40)
(not 10 < 20) or (30 < 20)
```

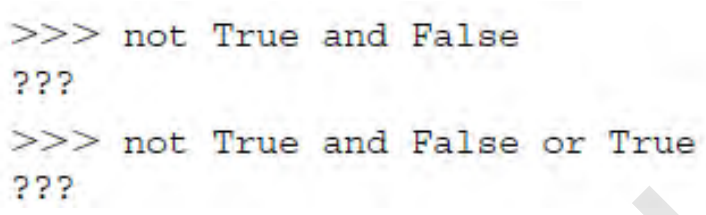
If not all subexpressions,



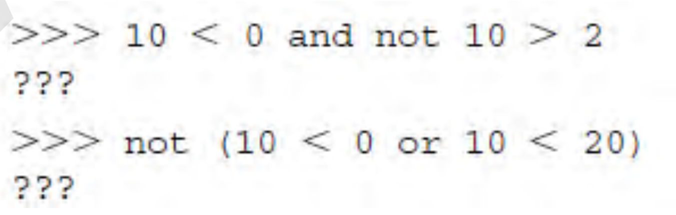
```
((10 < 20) and (30 < 20)) or (30 < 40)
(not (10 < 20)) or (30 < 20)
```

Let’s Try It

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



```
>>> not True and False
???
>>> not True and False or True
???
```



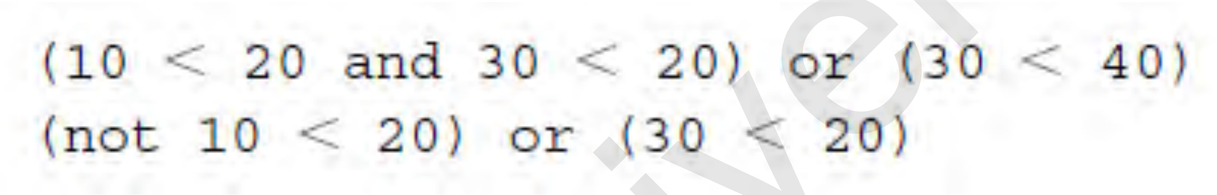
```
>>> 10 < 0 and not 10 > 2
???
>>> not (10 < 0 or 10 < 20)
???
```

False

True

False

False



```
(10 < 20 and 30 < 20) or (30 < 40)
(not 10 < 20) or (30 < 20)
```

True

False

Short-Circuit Evaluation

There are differences in how Boolean expressions are evaluated in different programming languages. For logical and, if the first operand evaluates to false, then regardless of the value of the second operand, the expression is false. Similarly, for logical or, if the first operand evaluates to true, regardless of the value of the second operand, the expression is true.

Because of this, some programming languages do not evaluate the second operand when the result is known by the first operand alone, called short-circuit (lazy) evaluation.

Subtle errors can result if the programmer is not aware of this. For example, the expression

```
if n != 0 and 1/n < tolerance:
```

would evaluate without error for all values of n when short-circuit evaluation is used. If programming in a language not using short-circuit evaluation, however, a “divide by zero” error would result when n is equal to 0. In such cases, the proper construction would be,

```
if n != 0:
    if 1/n < tolerance:
```

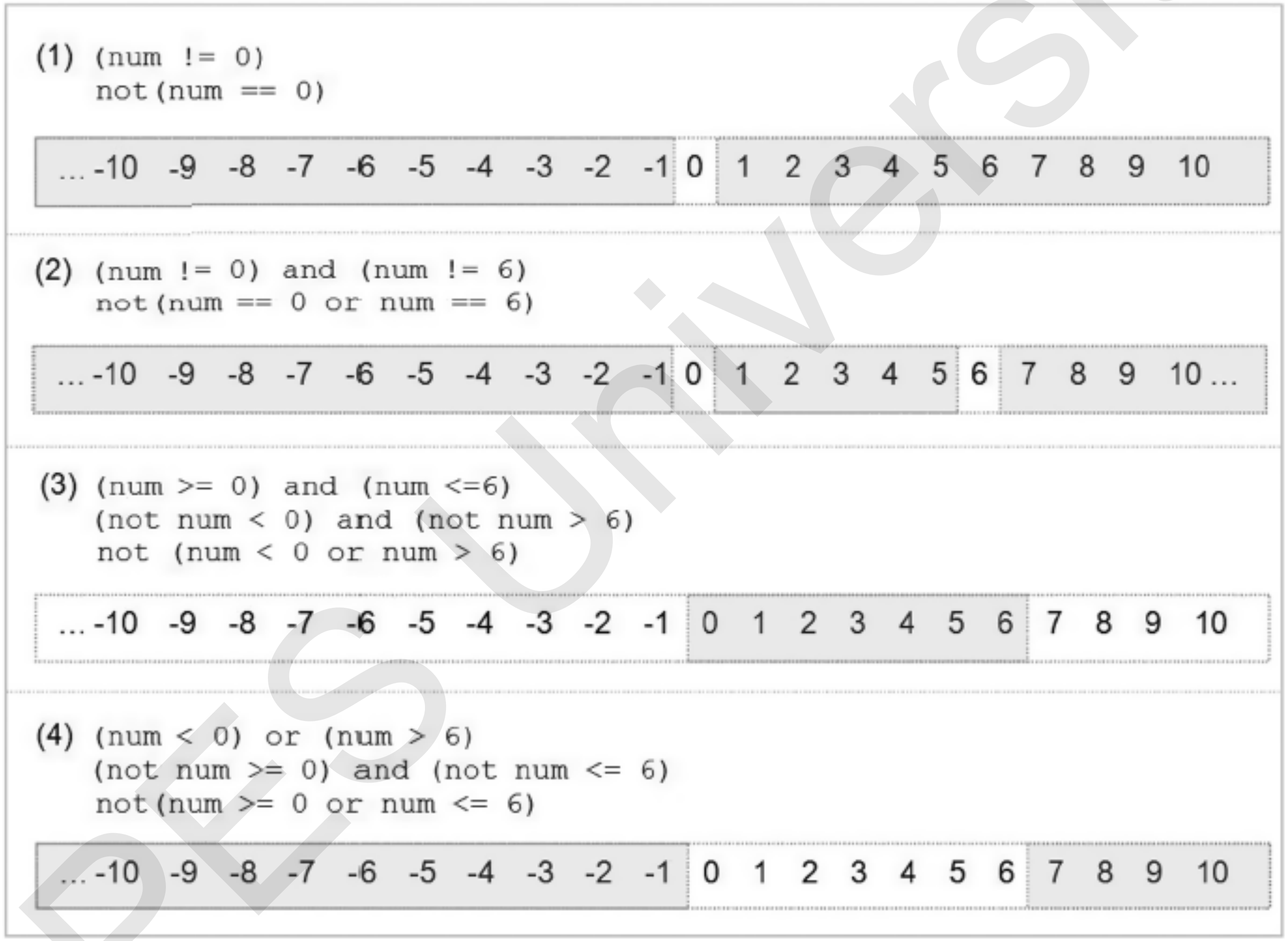
In the Python programming language, short-circuit evaluation is used.

Logically Equivalent Boolean Expressions

In numerical algebra, there are arithmetically equivalent expressions of different form.

For example, x(y + z) and xy + xz are equivalent for any numerical values x, y, and z. Similarly, there are *logically equivalent* ***Boolean expressions*** *of different form.*

Logically equivalent Boolean expressions of different form.



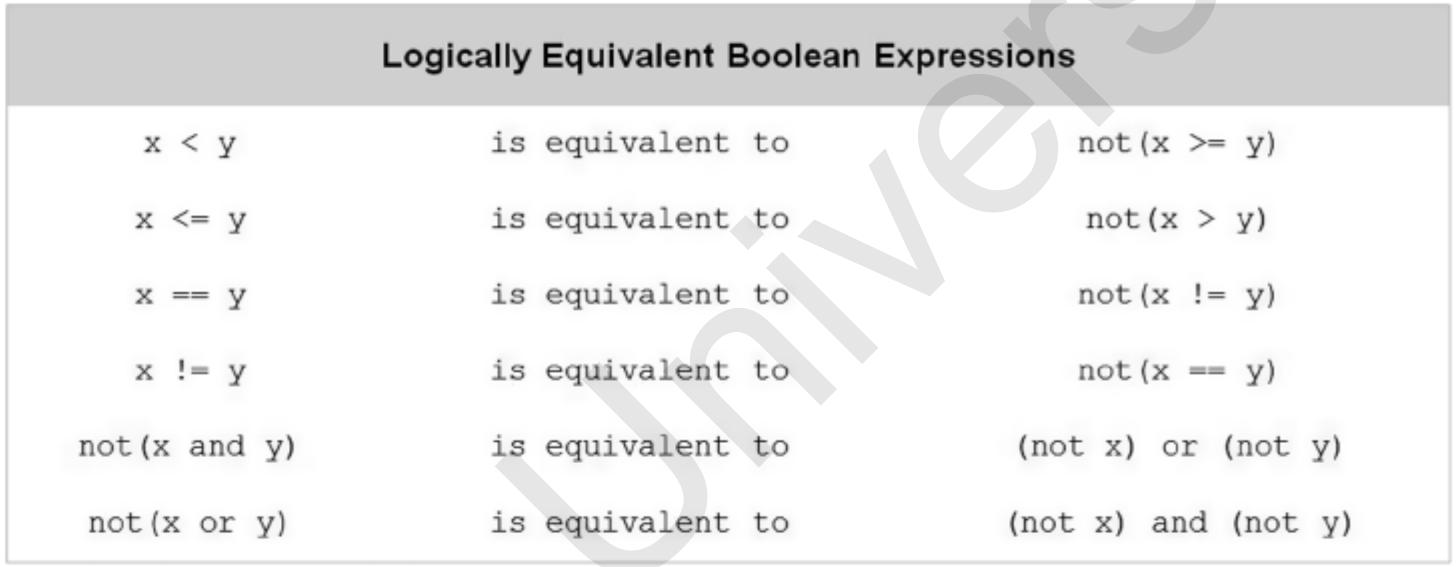
(1)
```
(num != 0)
not(num == 0)
```

(2)
```
(num != 0) and (num != 6)
not(num == 0 or num == 6)
```

(3)
```
(num >= 0) and (num <=6)
(not num < 0) and (not num > 6)
not (num < 0 or num > 6)
```

(4)
```
(num < 0) or (num > 6)
(not num >= 0) and (not num <= 6)
not(num >= 0 or num <= 6)
```

Fundamental logically equivalent Boolean expressions.

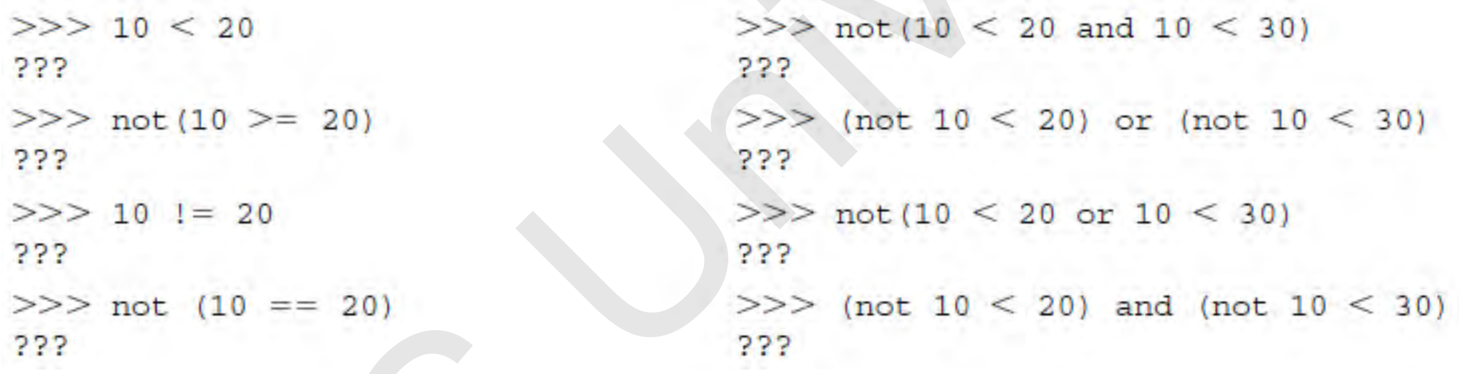


Logically Equivalent Boolean Expressions		
`x < y`	is equivalent to	`not(x >= y)`
`x <= y`	is equivalent to	`not(x > y)`
`x == y`	is equivalent to	`not(x != y)`
`x != y`	is equivalent to	`not(x == y)`
`not(x and y)`	is equivalent to	`(not x) or (not y)`
`not(x or y)`	is equivalent to	`(not x) and (not y)`

The last two equivalences are referred to as De Morgan's Laws.

Let’s Try It

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



```
>>> 10 < 20
???
>>> not(10 >= 20)
???
>>> 10 != 20
???
>>> not (10 == 20)
???
```

```
>>> not(10 < 20 and 10 < 30)
???
>>> (not 10 < 20) or (not 10 < 30)
???
>>> not(10 < 20 or 10 < 30)
???
>>> (not 10 < 20) and (not 10 < 30)
???
```

True
True
True
True

False
False
False
False