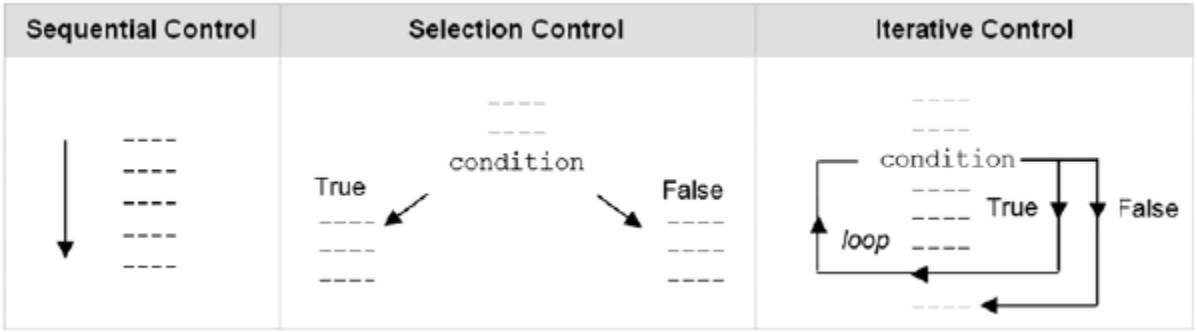
Boolean Expressions (Conditions)

What is a Control Structure?

Control flow is the order that instructions are executed in a program. A **control statement** is a statement that determines the control flow of a set of instructions. There are three fundamental forms of control that programming languages provide—sequential control, selection control, and iterative control.

Sequential control is an implicit form of control in which instructions are executed in the order that they are written. A program consisting of only sequential control is referred to as a "straight-line program." The program examples in Unit 2 are all straight-line programs. **Selection control** is provided by a control statement that selectively executes instructions, while iterative control is provided by an **iterative control** statement that repeatedly executes instructions. Each is based on a given condition. Collectively a set of instructions and the control statements controlling their execution is called a **control structure**.

Few programs are straight-line programs. Most use all three forms of control, depicted below:



Part I - Boolean Expressions (Conditions)

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, shown below. Relational expressions are a type of Boolean expression, since they evaluate to a Boolean result. These operators not only apply to numeric values, but to any set of values that has an ordering, such as strings.

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 the use of the **comparison operator**, `==`, for determining if two values are equal. This, rather than the (single) equal sign, `=`, is used since the equal sign is used as the assignment operator. This is often a source of confusion for new programmers,

`num = 10` variable `num` is assigned the value `10`

`num == 10` variable `num` is compared to the value `10`

Also note, **`!=`** is used for inequality simply because there is no keyboard character for the ≠symbol.

String values are ordered based on their character encoding, which normally follows a **lexographical** (**dictionary**) **ordering**. For example, `'Adam'` is less than `'Brenda'` since the Unicode (ASCII) value for `'A'` is 65, and `'B'` is 66. However, `'adam'` is greater than (comes *after*) `'Brenda'` since the Unicode encoding of lowercase letters (97, 98, . . .) comes *after* the encoding of uppercase letters (65, 66, . . .).

(Remember that the encoding of any character can be obtained by use of the `ord` function.)

Your Turn

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

```
>>> 15 = = 23                    >>> '13' , '29'
???                               ???
>>> 15 != 23                     >>> '13' < '9'
???                              ???
>>> 15 <= 23                     >>>  '13' > '7'
???                               ???
>>> 'Hello' = = 'Hello'         >>> 'Hello' < 'Zebra'
???                               ???
```

Part II - 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. The membership operators are given in Figure 3-4.

The `in` operator is used to determine if a specific value is in a given list, returning `True` if found, and `False` otherwise. The `not in` operator returns the opposite result. The list of values surrounded by matching parentheses in the figure are called tuples in Python.

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.

Your Turn

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

```
>>> 15 in (50, 20, 15)
???

>>> 15 not in (50, 20, 15)
???

>>> .33 in (.24, .37, .56)
???
```

```
>>> grade = 'B'
>>> grade in ('A','B','C','D','F')
???

>>> city = 'Hartford'
>>> city in ('Boston','Chicago', 'NY')
???

>>> "art" in "heart"
???
```

Part III - 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` in Python. These logical operators can be used to construct arbitrarily complex Boolean expressions. The Boolean operators are shown below

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`.

One must be cautious when using Boolean operators. For example, in mathematics, to denote that a value is within a certain range is written as

```
1 <= num <= 10
```

In most programming languages, however, this expression does not make sense. To see why, let's assume that `num` has the value `15`. The expression would then be evaluated as follows,

```
1 <= num <= 10      →     1 <= 15 <= 10     →      ?!?
```

It does not 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.) The correct way of denoting the condition is by use of the Boolean `and` operator,

```
1 <= num and num <= 10
```

In some languages (such as Python), Boolean values `True` and `False` have integer values 1 and 0, respectively. In such cases, the expression `1 <= num <= 10` would evaluate to `True <= 10` would evaluate to `1 <= 10`, which equals `True`. This would not be the correct result for this expression, however. Let's see what we get when we do evaluate this expression in the Python shell,

```
>>>num = 15
>>>1 <= num <= 10
False
```

We actually get the correct result, `False`. So what is going on here? The answer is that Python is playing a trick here. For Boolean expressions of the particular form,

```
value1 <= var <= value2
```

Python automatically rewrites this before performing the evaluation,

```
value1 <= var and var <= value2
```

Thus, it is important to note that expressions of this form are handled in a special way in Python, and would not be proper to use in most other programming languages.

One must also be careful in the use of `and/or` Boolean operators. For example, `not(num == 0 and num == 1)` is `True` for any value of `num`, as is `(num != 0) or (num != 1)`, and therefore are not useful expressions. The Boolean expression `num < 0 and num > 10` is also useless since it is always `False`.

Finally, Boolean literals `True` and `False` are never quoted. Doing so would cause them to be taken as string values (`'True'`). And as we saw, Boolean expressions do not necessarily

contain Boolean operators. For example, `10 < = 20` is a Boolean expression. By definition, Boolean literals `True` and `False` are Boolean expressions as well.

Your Turn

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

```
>>> True and False
???

>>>True or False
???

>>>not(True) and False
???

>>> not(True and False)
???

>>>(12 < 5) and (12 < 3)
???

>>>(12 < 5) or (12 > 3)
???

>>>not(12 < 5) or (12 > 3)
???

>>> not(12 < 5 or 12 > 3)
???
```

Part IV - Operator Precedence and Boolean Expressions

Operator precedence also applies to Boolean operators. Since Boolean expressions can contain arithmetic as well as relational and Boolean operators, the precedence of all operators needs to be collectively applied. An updated operator precedence table below.

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 before, in the table, higher-priority operators are placed above lower-priority ones. Thus, we see that *all arithmetic operators are performed before any relational or Boolean operator*,

```
10 + 20 < 20 + 30  →   30 < 50 → True
```

In addition, *all of the relational operators are performed before any Boolean operator*,

```
10 < 20 and 30 < 20 →  True and False → False

10 < 20 or 30 < 20  →  True or False → True
```

And as with arithmetic operators, Boolean operators have various levels of precedence. Unary Boolean operator `not` has higher precedence than `and`, and Boolean operator `and` has higher precedence than `or`.

```
10 < 20 and 30 < 20 or 30 < 40 → True and False or True
                               → False or True → True

not 10 < 20 or 30 < 20 → not True or False
                       → False or False → False
```

As with arithmetic expressions, it is good programming practice to use parentheses, even if not needed, to add clarity and enhance readability. Thus, the above expressions would be better written by denoting at least some of the subexpressions,

```
(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)
```

Finally, note from the table above that all relational and Boolean operators associate from left to right.

Your Turn

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

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

>>> not(True and False or True
???

>>> 12 < 5) and (12 > 3)
???

>>> not(12 < 5 or 12 > 30)
???
```

Part V - More Boolean Expressions

Short-Circuit (Lazy) 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

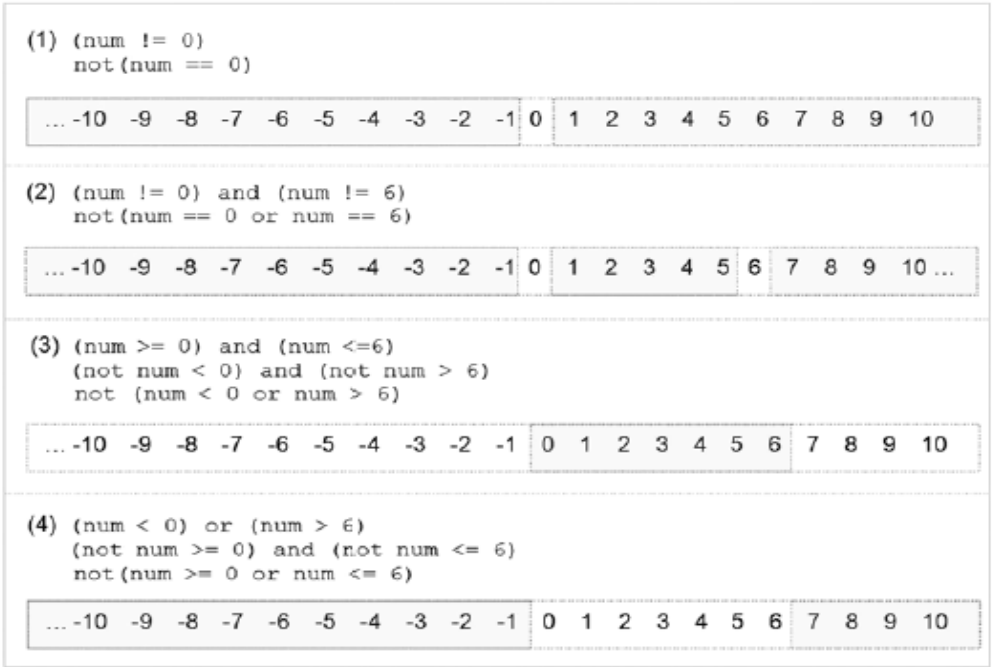
```
if n != 0 and 1/n < tolerance:
```

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:
```

Logically Equivalent Boolean Expressions

In numerical algebra, there are arithmetically equivalent expressions of different form. For example, `x(y 1 z)` and `xy 1 xz` are equivalent for any numerical values `x`, `y`, and `z`. Similarly, there are *logically* equivalent **Boolean expressions** of different form. Some examples are below:



The range of values satisfying each set of expressions is shaded in the figure. Both expressions in (1) are true for any value except 0. The expressions in (2) are true for any value except 0 and 6. The expressions in (3) are only true for values in the range 0 through 6, inclusive. The expressions in (4) are true for all values except 0 through 6, inclusive. The table below lists common forms of logically equivalent 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 above are referred to as De Morgan's Laws.

Concepts and Procedures

1. Which of the three forms of control is an implicit form of control?

2. What is meant by a "straight-line" program?

3. What is the difference between a control statement and a control structure?

4. Three forms of control in programming are sequential, selection, and _____ control.

5. Which of the following expressions evaluate to `True`?

a) `10 <= 8`
b) `8 <= 10`
c) `10 == 8`
d) `10 != 8`
e) `'8' < '10'`

6. Which of the following Boolean expressions evaluate to `True`?

a) `'Dave' < 'Ed'`

b) `'dave' < 'Ed'`
c) `'Dave' < 'Dale'`

7. What is the value of variable `num` after the following is executed?

a) `<<< num = 10`
b) `<<<num = num + 5`
c) `<<<num = = 20`
d) `<<<num = num + 1`

8. What does the following expression evaluate to for `name` equal to `'Ann'`?

```
name in ('Jacob', 'MaryAnn', 'Thomas')
```

9. Evaluate the following Boolean expressions using the operator precedence rules of Python.

a) `10 <= 8 and 5 != 3`
b) `10 <= 8 and 5 = = 3 or 14 < 5`

10. Which one of the following Boolean expressions is not logically equivalent to the other two?

a) `not(num < 0 or num > 10)`
b) `num > 0 and num < 10`
c) `num <= 0 and num <= 10`

Problem Solving

1. Write a Python program in which the user enters either 'A', 'B', or 'C'. If 'A' is entered, the program should display the word 'Apple'; if 'B' is entered, it displays 'Banana'; and if 'C' is entered, it displays 'Coconut'. Use nested if statements for this.

2. Repeat question 1 using an `if` statement with `elif` headers instead.