**1.3.2 Boolean**

***Boolean***, or ***relational***, expressions are expressions that conceptually result in the values true or false. The type for these expressions is **bool**. There are built-in values **True** and **False** (note that these are capitalized). Expressions that result in **True** are equivalent to the integer 1, and expressions that result in **False** are equivalent to the integer 0. The binary ***relational operators***

that can be used in these expressions include:

< less than

> greater than

<= less than or equals

>= greater than or equals

== equality

!= inequality

Boolean expressions return the value of **True** or **False**.

*>>> 3 < 4*

True

*>>> 6 == 5*

False

Using the integer remainder division, we can write an expression to test whether an integer is even or odd. If an integer is even, it will be divisible by 2, so the remainder division operator would return 0. For example, 8%2 is 0, so 8%2 == 0 would be **True**. For a variable *var*, the expression var%2==0 would be **True** if the variable is even, or **False** if it is odd.

*>>> var = 16*

*>>> var % 2 == 0*

True

*>>> var = 9*

*>>> var % 2 == 0*

False

Math can be done on the result of Boolean expressions, since **True** is the same as 1 and **False** is the same as 0.

*>>> myval = 3 < 4*

*>>> myval + 7*

8

The ***logical operators*** operate on Boolean expressions; they are **not**, **and**, and **or**. The **not** operator is unary; it results in the opposite of the Boolean expression. For example, **not True** is **False**, and **not False** is **True**. The **and** and **or** operators are binary. The **or** operator returns **True**

if either or both operands are **True**. The **and** operator only returns **True** if both operands are **True**.

*>>> 3 < 7 or 2 == 4*

True

*>>> 3 < 7 and 2 == 4*

False

*>>> 3 < 7 and 2 == (3 – 1)*

True

Representing the concepts of true and false is somewhat different. For numbers, 0 represents the concept of false, but anything that is not 0 can be used to represent the concept of true. We will see examples and applications of this in later sections and chapters.

**1.3.3 Precedence Table**

Expressions in parentheses take ***precedence***, or ***priority***, over the operators. There are also precedence rules for the operators themselves, e.g., multiplication takes precedence over addition.

For the math and Boolean operators that have been covered so far, the following shows the operator precedence rules. Operators on each line have priority over the operators below them in the table.

Parentheses ( )

Exponentiation \*\*

Negation -

Multiplication and Division \*, /, //, %

Addition and Subtraction +, -

Relational <, <=, >, >=, ==, != **not not**

**and and**

**or or**

When operators have the same precedence, they are evaluated from left to right. This is called the ***associativity***.

**1.3.4 Introduction to Sequences**

Numbers and Boolean expressions are simple types that only store one value. ***Data structures*** are structured variables that store more than one value. In this section, we will briefly introduce two types of data structures, ***strings*** and ***lists***. Both of these are particular kinds of data structures called ***sequences***. With sequences, the values are put in an order and can be indexed using integer indices. Much more information on strings and lists will be covered in Chapters 5 and 7.

**1.3.4.1 Strings**

Strings in Python are text, and can be stored in either single quotes (‘Hello’) or double quotes (“Hi!”). A string is a sequence of characters. For example, the following code stores strings in variables and then prints them.

*>>> stringone = 'Hello there'*

*>>> string2 = "!!!"*

*>>> print(stringone, string2)*

Hello there !!!

Printing strings using the **print** function will not show the quotes. However, displaying the value of a string variable will. Note that although the variable *string2* is created using double quotes, the default is to display the value of a string with single quotes.

*>>> string2*

'!!!'

The type name for strings is **str**.

In some languages there is a distinction between a single character and a string of characters, but in Python they are all strings. A single character is just a string that has only one character in it.

Strings are composed of individual characters. The individual characters are numbered with integers beginning with 0. These numbers are called ***subscripts*** or ***indices***. For example, for the variable *stringone*, the 11 characters have indices from 0 through 10. In the diagram, the indices are shown above the characters in the string.

0 1 2 3 4 5 6 7 8 9 10

H e l l o t h e r e

An individual character can be obtained by putting the index in square brackets after the string variable name. This is called ***indexing*** into the string. For example, the following will print the first two characters in the string variable *stringone*.

*>>> print(stringone[0], stringone[1])*

H e

The **len** function will return the length of the string, which is the number of characters in a string, e.g.:

*>>> len(stringone)*

11

With a length of 11, the string has indices from 0 through 10.

For strings, indexing returns the individual characters in the string.

The ***empty string*** is a string that contains zero characters. An empty string can be created using quotes with nothing inside, e.g. using single quotes:

*>>> len('')*

0

The **in** operator will determine whether or not a string is in another string, and return **True** if it is or **False** if it is not. The **not in** operator will return the opposite.

*>>> 'x' in 'abcde'*

False

*>>> 'x' not in 'abcde'*

True

Strings are ***immutable***, which means that they cannot be modified. Any attempt to change character(s) in a string will result in an error message.

*>>> myword = 'Hello'*

*>>> myword[1] = 'a'*

TypeError: 'str' object does not support item assignment

There are operators for strings. To ***concatenate***, or join strings together, the ‘+’ operator is used.

*>>> 'b' + 'cde'*

'bcde'

Note that there are no spaces in between the concatenated characters.

Strings can be compared using the relational operators. String comparisons are based on the ***character encoding***. Two common encoding sequences are ASCII and Unicode. Basically, characters are put into a sequence and given equivalent integer values. In most encoding sequences the letters of the alphabet are ordered sequentially, so it is true that ‘a’ is less than ‘c’ because ‘a’ comes before ‘c’ in the encoding sequence.

*>>> 'a' < 'c'*

True

If one string is shorter than another, but they start with the same characters, the longer string is greater than the shorter string.

*>>> 'ab' < 'abc'*

True

The **print** function can print combinations of strings and numbers, e.g.:

*>>> string2 = "!!!"*

*>>> print("The number is", 33, string2)*

The number is 33 !!!

Note that blanks are automatically inserted in between the text and numbers that are printed. As we have seen, by default the contents of the strings are printed without quotes around them.

To print a string variable with quotes around it, the function **repr** can be used. This function returns the string representation of the variable (again, always in single quotes).

*>>> word = "Monty"*

*>>> print(word, repr(word))*

Monty 'Monty'

**1.3.4.1 Lists**

Lists are sequences of values. Simple lists are created by putting values in square brackets, separated by commas.

The values in lists can be different types, although it is more common for them to be the same type.

*>>> numlist = [4, 52, 33, 11, -3]*

*>>> print(numlist, type(numlist))*

[4, 52, 33, 11, -3] <class 'list'>

*>>> mixedlist = [22, 'hello', 4 > 6]*

*>>> print(mixedlist, type(mixedlist))*

[22, 'hello', False] <class 'list'>

Notice that the type of any list is **list**. Note also that the result of the expression 4>6, **False**, is stored in the list.

Since lists are a sequence type, they can be indexed using integers.. However, unlike strings, lists are a ***mutable*** type, which means that they can be modified. The values stored in lists are called ***items*** or sometimes ***elements***. The following creates a list and then indexes into the list to get the first value (which is in element 0).

*>>> numlist = [4, 52, 33, 11, -3]*

*>>> numlist[0]*

4

Since lists are mutable, items can be modified by indexing and assigning a new value.

*>>> numlist = [4, 52, 33, 11, -3]*

*>>> numlist[1] = 99*

*>>> print(numlist)*

[4, 99, 33, 11, -3]

Lists can contain other sequence types, for example, strings.

*>>> strlist = ['hi', 'hello', 'ciao']*

*>>> strlist[0] = 'howdy'*

*>>> print(strlist)*

['howdy', 'hello', 'ciao']

The **len** function will return the number of elements in a list. There are 3 items in the variable *strlist*. The third item in the list (strlist[2]) is a string that has 4 elements, or characters.

*>>> len(strlist)*

3

*>>> len(strlist[2])*

4

The **in** operator will determine whether a value is in a list or not.

*>>> 4 in numlist*

True

*>>>* '*hello*' *in strlist*

True

*>>>* '*e*' *in strlist*

False

The character 'e' is not one of the elements of *strlist*, although it is in strlist[1].

**1.3.5 Type Casting**

***Type casting*** means converting an expression from one type to another. The type names in Python can be used as function names to accomplish this. For example, to convert *x* to a **float**, use **float(x)**, e.g.

*>>> float(4)*

4.0

Note that in the following, the // operator still returns the result of the integer whole number division; the integer result of 3 is then converted to the type **float**:

*>>> float(7//2)*

3.0

Converting a **float** to an **int** will truncate the decimal places.

*>>> int(5.7)*

5

A string variable that contains a number can be converted to a number type, e.g.:

*>>> strnum = '123'*

*>>> actualnum = int(strnum)*

*>>> print(actualnum, type(actualnum))*

123 <class 'int'>

Conversely, a number can be converted to a string. For example, if we wanted to know how many digits in a number, we could convert the number to a string and then use the **len** function (which does not work on numbers) to get the number of digits:

*>>> mynum = 1234*

*>>> strnum = str(mynum)*

*>>> print(len(strnum))*

4

Using the **bool** function to cast numbers to the type **bool**, we can see that numbers that are not 0 can be used to represent the concept of true.

*>>> print(bool(0), bool(1), bool(33), bool(4.2))*

False True True True