**PYTHON DAY 3**

**DATA TYPES**

Data types covered in this course include:

* Integers
* Booleans
* Floating point numbers
* Complex numbers
* Strings

Literals are an alternative to using variables. Examples of literals include:

* "This is only a string"
* "\t"
* 2

Literals are values that never change, i.e. they are a constant. Look at the following example:

>>>             #Interpreter

>>> print ("The answer: ", 5 + 2)

>>> The answer: 7

>>>

5 and 2 are constant values; they are not assigned to a variable, and therefore are literal values.

**NOTE:** You do not have to convert 5 + 2 to a string because the two integer values are added and automatically converted because explicit casting is not required in this case.

**Data Types**

**Data Types**

**Examples**

 Integers

 These represent numbers in an unlimited range. This is only limited by a machine’s memory.

 Booleans

 Evaluate to ‘True or False’, 1 or 0 respectively.

 Floating point numbers

 Floating-point numbers represent double-precision numbers.

 Complex numbers

 Complex numbers represent numbers as a pair of double-precision numbers.

 Strings

 A sequence of Unicode characters e.g. a word or a sentence that can be manipulated.

**INTEGERS**

We say that Python is a static language because only values of a certain type can be assigned to particular variables, according to their data types assigned.

Integers are always whole numbers. Integers include negative and positive numbers. The only factor that determines the range of an integer variable is the amount of memory a machine has available.

**BOOLEANS**

Boolean data type has corresponding integer values. There are only two possible values that a Boolean variable can have, **True** (1) or **False** (0). When returning Booleans as strings they are seen as “True” and “False”, and never as “1” and “0”. True and False are case-sensitive in Python. Boolean tests whether conditions are valid or not. The three logical operators used to test conditions between two arguments are:

* The **and-operator**
* The **or** operator
* The **not** operator

The following variable values are considered **False**:

* False
* None
* Zero for any numeric data type, **0, 0.0, 0j**
* An empty sequence or mapping. Like a list or tuple, **' '**, **( )**, **[ ]**, **{ }**
* Instances of user-defined classes, where a class that defines a**\_\_bool\_\_()** method returns **zero** or **False**.

All values returned otherwise are always considered true. This means that many objects will always return true.

Operators and built-in functions that have a Boolean result always return (False or 0) or (True or 1). The Boolean **or** and **and** operations always return only one of the options, either **True** or **False**.

**Manipulate Booleans**

**Syntax**

**Description**

 a or b

If either a or b is True, then the result will be True.

If both a and b are False then the result will be False.

 a and b

 If a and b are True, then the result will be True. Otherwise, the result will be False.

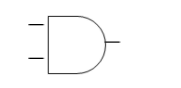
 not a

 If a is True, False is returned. If a is False, True is returned.

Electronic diodes work on the following concept:

Diodes can be in two states. The one state is **on**, and the other is **off**. This also means **True** and **False** respectively. Diodes have two inputs to verify what the output should be.

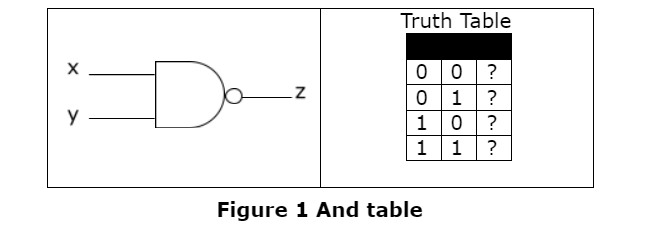
* The **AND** gate: Both inputs have to be **on** in order for the gate to have an **on** output.



* **NOT**: Converts **True** to **False**, and **False** to **True**.

image 2

Write a program that will fill in the following truth table for the example that follows:



**Syntax**

**Description**

 a or b

If either a or b is True, then the result will be True.

If both a and b are False then the result will be False.

 a and b

 If a and b are True, then the result will be True. Otherwise, the result will be False.

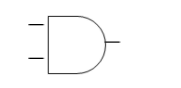
 not a

 If a is True, False is returned. If a is False, True is returned.

Electronic diodes work on the following concept:

Diodes can be in two states. The one state is **on**, and the other is **off**. This also means **True** and **False** respectively. Diodes have two inputs to verify what the output should be.

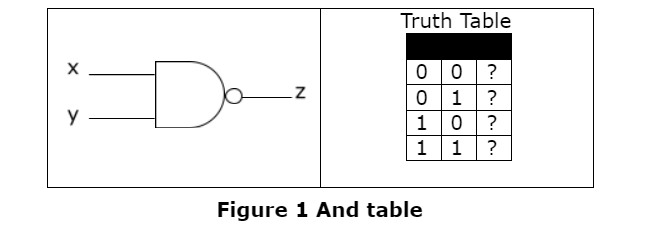
* The **AND** gate: Both inputs have to be **on** in order for the gate to have an **on** output.



* **NOT**: Converts **True** to **False**, and **False** to **True**.

image 2

Write a program that will fill in the following truth table for the example that follows:



**Example 2: AND gate:**

1 x = bool()

2 y = bool()

3

4 print ('Enter x as 1 or 0:')

5 x = int(input())

6

7 print ('Enter y as 1 or 0:')

8 y = int(input())

9

10 z = str(not bool(x and y))

11

12 print ('The Boolean value of x is', str(bool(x)))

13 print ('The Boolean value of y is', str(bool(y)))

14 print ('The Boolean value of (x and y) is', str(bool(x and y)))

15 print ('The Boolean value of (x not y) is', z)

**Output:**

>>> ==================== RESTART ================================

>>>

Enter x as 1 or 0:

0

Enter y as 1 or 0:

0

The Boolean value of x is False

The Boolean value of y is False

The Boolean value of (x and y) is False

The Boolean value of (x not y) is True

>>> ==================== RESTART ================================

>>>

Enter x as 1 or 0:

0

Enter y as 1 or 0:

1

The Boolean value of x is False

The Boolean value of y is True

The Boolean value of (x and y) is False

The Boolean value of (x not y) is True

>>> ==================== RESTART ================================

>>>

Enter x as 1 or 0:

1

Enter y as 1 or 0:

0

The Boolean value of x is True

The Boolean value of y is False

The Boolean value of (x and y) is False

The Boolean value of (x not y) is True

>>> ===================== RESTART ================================

>>>

Enter x as 1 or 0:

1

Enter y as 1 or 0:

1

The Boolean value of x is True

The Boolean value of y is True

The Boolean value of (x and y) is True

The Boolean value of (x not y) is False

>>>

Lines 1 and 2 define **x** and **y** as Boolean values. Lines 4 to 8 get input from the user. Line 10 calculates **z**, which is inverted because the **AND** gate has a **NOT** gate attached to it, as seen in Figure 1.

z = str(not bool(x and y))

First of all x and y are compared to check if they are the same x and y. If they are, 1 is returned, otherwise, **0** is returned. bool(x and y): The value is then converted from the number form (**0** or **1**) to the word form of Boolean (True or False). not bool(x and y) converts **True** to **False**, or False to True. z is then converted to a string.

Lines 12 to 15 are formatted print statements.

print ('The Boolean value of x is', str(bool(x)))

print ('The Boolean value of y is', str(bool(y)))

print ('The Boolean value of (x and y) is', str(bool(x and y)))

print ('The Boolean value of (x not y) is', z)

**FLOATING POINT NUMBERS**

Floating point numbers are better known as **floats**. Float is the data type that manages numbers with decimal places with very accurate precision. The float data type can be called as a function with zero or 1 argument of any data type. If no argument is given, then float returns 0.0. If an argument is given, an attempt will then be made to convert the value to a float data type, but this does not mean it is always possible. For example, float("21.765") will be converted to a float, but float("FF909A") will raise an exception.

A string value cast to a float must contain only numbers and only one occurrence of the dot (.) character. The following piece of code will clarify the statement:

The following example shows the different ways to format float numbers:

**Example 3: Format strings:**

>>>             #Interpreter

>>> print ("Today's Dollar price compared to the Rand: R%f" % 6.85871)

Today's Dollar price compared to the Rand: R6.858710

>>>

>>> print ("Today's Dollar price compared to the Rand: R%.2f" % 6.85871)

Today's Dollar price compared to the Rand: R6.86

>>>

>>> print ("Change since yesterday: R%+.2f" % 0.5)

Change since yesterday: R+0.50

>>>

The example compares the American dollar to the South African rand over two days; the formats of the relevant values are changed to only print two decimal values.

The f in the format part of the print statement indicates that the number to be formatted is a float.

The + sign indicates that the changed amount must be signed, which means that if the dollar compared to the rand decreases, that a − sign would be printed before the amount difference.

The following examples should clear any misunderstanding:

>>>             #Interpreter

>>> print ("Negative amount: %+f" % -0.23)

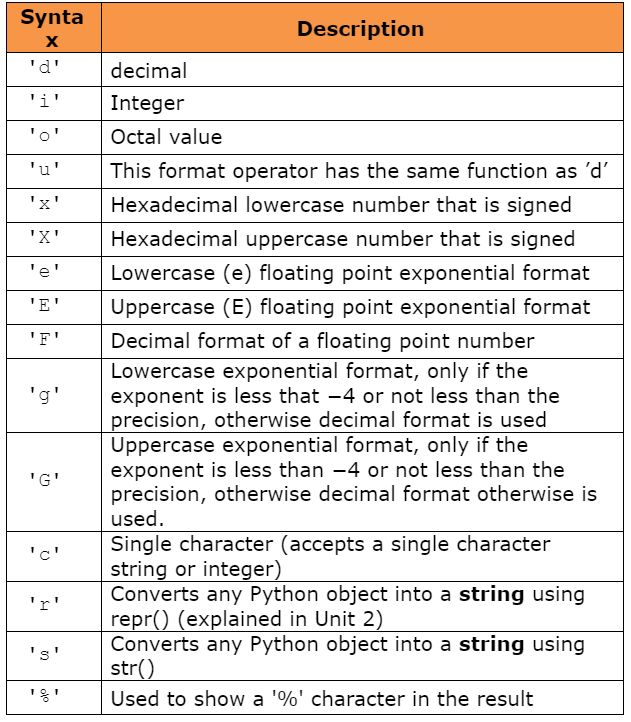
Negative amount: -0.230000

>>> print ("Positive amount: %+f" % 0.23)

Positive amount: +0.230000

The following table shows tokens that can be used to format a string:

**Manipulate strings**



**COMPLEX NUMBERS**

Complex numbers are two numbers contained in a single variable. The first part of a complex number is the real part (float), and the second part is the imaginary part (float), assigned in this manner: complex(real, image). Imaginary numbers are real multiples of the imaginary unit, written with a suffix of **j (J)**. The imaginary part is **the square root of -1**. Python has built-in support for complex numbers. The latter notation is written as follows: 4+8j.

Complex numbers are used in Python to combine two numbers into one manageable number. The following example shows how complex numbers are used in a simple manner.

**Example 5: Complex numbers:**

1  a\_comp = complex(4, 8) # defines a\_comp complex number

2  i1 = 5

3  i2 = 6

4  b\_comp = complex(i1, i2) # complex(5, 6)

5

6  print (a\_comp.real) # first number which is 4

7  print (a\_comp.imag) # second number which is 8

8

9  print (abs(a\_comp)) # sqrt(a\_comp.real\*\*2 + a\_comp.imag\*\*2)

10 print (a\_comp + 5+6j) # adds a\_comp complex number to a

11 # prints the total

12 print (7+5j + 10+4j) # adds 7 + 10 and 5 + 4 separately

**Output:**

>>>

4.0

8.0

8.94427191

(9+14j)

(17+9j)

>>>

The absolute value of a complex number is its magnitude (or modulus), defined as the theoretical distance between the coordinates (real,imag) of x and (0,0) (applying the Pythagorean theorem). Another way of thinking about it is that it can be used to calculate the distance from 0, if you walk 5 km left and then 10 km up: you will be 11.19 km from where you started, because Banner Complete = 11.185.

In the first line, a **complex** number is defined, the **real** number is defined as **4** and the **imaginary** number as **8**. In lines 2 and 3, two integers are declared. Remember that integers can **implicitly** be converted to floats and that is why line 4 will execute with no exceptions (complex numbers accept two floating-point numbers). Lines 6 and 7 print the first and second numbers of a comp respectively. Lines 10 and 12 show that it is possible to manipulate complex numbers; they also show that the real values are added separately from the imaginary numbers.

**STRINGS**

Strings are represented by the immutable (unchangeable) str data type. Strings are a sequence of Unicode characters which form a single manageable string. The str data type can be called to create a string; when there is no argument supplied, it returns an empty string. s = str("") is the same as s = str(), when an argument is passed to the string method that is not a string value, it is passed as a string representation of the type supplied: s = str(17.2354), is the same as s = str("17.2354"). The string function is often used to convert other data types to strings.

**Example 6: Converting to string:**

1 print ("Please enter your name:")

2 s\_name = input(str())

3 print ("Please enter your surname:")

4 s\_surname = input(str())

5 print ("Please enter your current age")

6 i\_age = input(int())

7

8 s = str(i\_age)

9

10 sentence = s\_name + " "

11 sentence += s\_surname + " is currently "

12 sentence += s + " year(s) old"

13

14 print (sentence)

This example is straightforward. The user is asked to enter three values: his or her name, surname, and current age. Then a message is printed with the person’s name and surname and age with a single print statement. Look at the output of the code below if the program seems unclear.

**Output:**

>>>

Please enter your name:

John

Please enter your surname:

Doe

Please enter your current age

034

John Doe is currently 34 year(s) old

>>>

The example above asks you to input information with the intention that the program will use the information to produce a suitable output. You are asked to enter your name first. Your name and surname are stored as objects of the string type.

The ‘0’ in front of 34 in the output indicates that an integer value is expected.

**Using the ‘+=‘ operator with strings**

The ‘+=‘ operator adds values to an existing variable. This is illustrated in the next example:

**Example 7: The ‘+=‘ operator with strings:**

sentence = "This sentence is way too long to fit on one line of"

sentence += " code and that is why I'm breaking this sentence down"

sentence += " into pieces."

print (sentence)

The ‘+=‘ operator’s functionality is not only limited to strings, it can be used with other data types, too. More operators will be covered on Day 4.

**Using the end of line escape sequence**

The end of line escape sequence (\) can also help to make code more readable, as the following example shows:

**Example 8: End of line escape sequence:**

>>>             #Interpreter

>>> sentence = "This sentence is way too long to fit on one line of" \

        " code, and that’s why I'm breaking this sentence up" \

        " into pieces."

>>> print (sentence)

**Enclose the expression in brackets**

The easiest way to write code in an easily readable format is by enclosing the expression in brackets:

**Example 9: Enclose the expression in brackets:**

sentence = ("This sentence is way too long to fit on one line of"

      " code, and that’s why I'm breaking this sentence up"

      " into pieces.")

print (sentence)

**LAMBDA EXPRESSIONS**

Small anonymous functions can be created with the lambda keyword. This function returns the sum of its two arguments: lambda a, b: a+b. Lambda functions can be used wherever function objects are required.

They are syntactically restricted to a single expression. Semantically, they are just syntactic sugar for a normal function definition. Like nested function definitions, lambda functions can reference variables from the containing scope:

>>>

>>> def make\_incrementor(n):

...  return lambda x: x + n

...

>>> f = make\_incrementor(42)

>>> f(0)

42

>>> f(1)

43

The above example uses a lambda expression to return a function. Another use is to pass a small function as an argument:

>>>

>>> pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]

>>> pairs.sort(key=lambda pair: pair[1])

>>> pairs

[(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]