In the following concept we will be covering:

Introduction to Python

- Variables and print statements
- Numbers
- Strings

Data Structures in Python

- Lists
- Dictionaries
- Tuples
- Mutable vs Immutable

Operations in Python

- What are the Operators?
- Arithmetic Operators
- Comparison Operators
- Logical Operators
- Assignment and Bitwise Operators
- Membership Operators
- Identity Operators

In the concept Handling Program Flow in Python we will be drilling down further into advanced python:

- Control Statements and Loops
- Files I/O
- Exceptions
- Functions
- Object-Oriented Programming

Advantages of Python

Python was created by Guido van Rossum and first released in 1991. The name Python was inspired by the comedy series Monty Python's Flying Circus. The following are some of the major advantages of using Python

Easy to Learn

Python is very beginner-friendly. The syntax (words and structure) is extremely simple to read and follow, most of which can be understood even if you do not know any programming.

Versatility

Python is a multi-paradigm programming language. It supports object-oriented programming, structured programming, and functional programming patterns, among others. Python can handle every job ranging from data mining to website construction to running embedded systems, all in one unified language.

Community Support

Python's broad and diverse base means that there are millions of users who are happy to offer advice or suggestions when you get stuck on something. Chances are, someone else has been stuck there first. Open-source communities are known for their open discussion policies. In fact, most of the queries can be solved with a simple *Google Search*.

Awesome and Free Libraries

The libraries form the backbone for the success of any programming language. At present more than 140,000 projects exist in the Python Package Index (PyPI).

Data Science has been an early beneficiary of these libraries. Libraries like NumPy, Pandas, and Scikit-learn are extremely popular and widely used by Data Scientists worldwide. You want to start doing Deep Learning. No problems! Python has libraries like Tensorflow, Pytorch, and Keras to help you.

Interpreted Language

Python is an interpreted language, not a compiled one. This statement means that the original program is translated into "something else". Another program -- "the interpreter" -- then examines "something else" and performs whatever actions are called for.

Besides, Python is dynamically typed, i.e., we don't need to define variable data type ahead of time; Python automatically guesses the data type of the variable based on the type of value it contains.

What are Variables?

Variables are used to store values in memory. We can store integers, decimals, characters, words or sentences in a variable. In order to assign value to a variable, use the equal(=) sign.

For example:

```
x = 1
y = 2.0
name =  "Paul"
```

Here, `1`, `2.0`, and "`Paul`" are assigned to the variables x, y, and name respectively.

Data Types in Python

Python has five standard data types, listed as follows:

- Boolean
- Numbers
- String
- List
- Dictionary

- Tuple

Boolean

Boolean is nothing but `True` (`1`) and `False` (`0`) statements. Their sole purpose is to evaluate conditions.

For example: Is 2 < 3? Yes, so it will return a value `1`.

Is 'abc' equal to 'abc' (written as `'abc' == 'abc'`? Yes, again.

We will take look at the other data types in upcoming topics

How to Use the `print()` Function

The `print()` function is used to display the variable's result.

The syntax is: `print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)`

Here,

- `objects` is/are the values to be displayed
- `sep` is the separator used between the values (defaults into a space character)
- After all the values are printed, `end` is printed (defaults into a new line)
- `file` is the object where the values are printed (default value is `sys.stdout` i.e. screen)

For example:

```
x = 1
print(x)
```

Output is:

```
1
```

Determining the Type of Object with `type()`

If you are confused about what class an object belongs to simply use the `type()` function. In the example given above, the types of the variables are `int`, `float` and `str` respectively.

For example:

```
print(type(x))
```

prints out `int`

Checking Instance Type with `isinstance()`

The `isinstance()` function checks if the object (first argument) is an instance or a subclass of classinfo (second argument).

Its syntax is: `isinstance(object, classinfo)`

It returns `True` if the object is an instance or subclass of a class or any element of the tuple `False` otherwise.

For example:

```
isinstance(x, int)
```

returns `True` while

```
isinstance(y, int)
```

returns `False`

Numeric Types

It is used to store numeric values. For example:

$x=5$

$x=5,$

$y = 10$

$y=10,$

$\ balance = 15{,}000$

$balance=15{,}000$ etc.

Python supports three different types of numbers, listed as follows:

- Int (integers)
- Float (decimal numbers)
- Complex (complex numbers)

Integers are positive or negative whole numbers with no decimal point. For example: `2`and `4` are integers.

Floats are numbers containing a decimal point (`.`). For example: `2.3`. Note that numbers written in a scientific notation such as

$\ 2.5*10\^5$

$2.5*10$

are float number because it contains a decimal point in the notation.

Python also provides support for complex numbers of the form

\ a + bJ

$a+bJ$, where `a` and `b` are floats and `J`'s value is the square root of minus 1. Its real part is `a` and the imaginary part is `b`.

Common Numeric Operations

Symbol	Function	Example
+	Addition	2 + 3 2+3
-	Subtraction	2 - 3 2–3
*	Multiplication	2*3 2*3
//	Integer division (returns only quotient)	3//2 3//2
/	Division	3/2 3/2
**	Exponentiation	2^3 2 3

abs(x)	Absolute value	|x| $\|\| x \|\|$
exp(x)	e raised to the power x	e^x e^x
log(x)	Logarithm with base e	log_e x $log_e x$
log10(x)	Logarithm with base 10	log_{10} x $log_{10} x$
pow(x,y)	x raised to the power y	x^y x^y

Type Conversion

During data manipulation, there might arise a need for converting an integer to float, float to an integer, float to a string, or splitting on a decimal point in a float. Let's look at how you can do this.

Let's say you have number `x`. The following are some common operations:

- int(x) converts it to an integer.
- float(x) converts it to a float.

- complex(x) converts it to a complex number with real part `x` and imaginary part `0`.
  *You can delete a number object `x` by simply typing `del x`.

Definition

A string is a collection of alphabets, words, or characters represented with the help of quotation marks. Python has a built-in class for `str` to handle string-related operations.

Representation

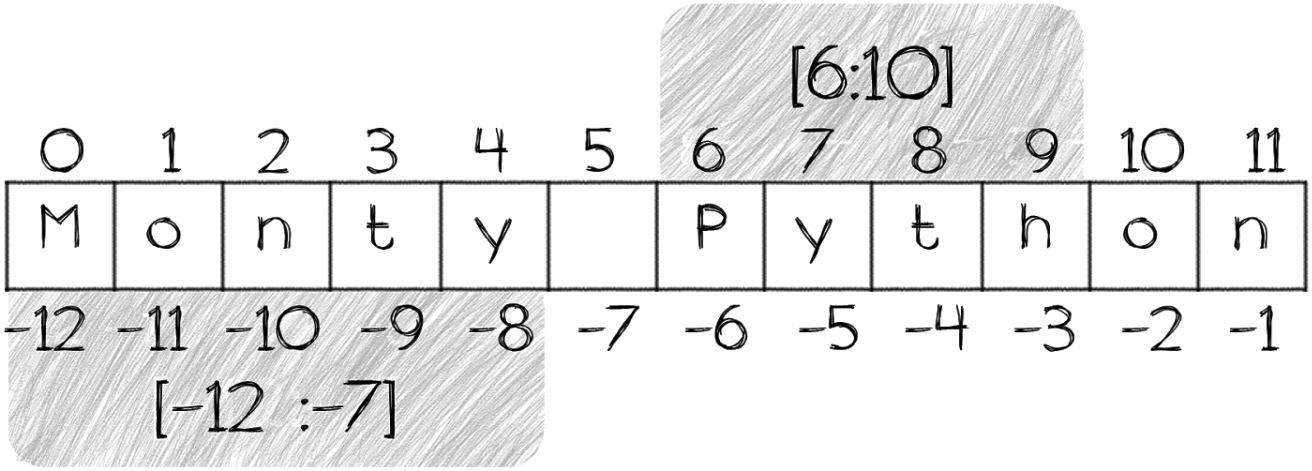
Wrap an object within quotes to denote it as a string. The possible ways to do this are:

- Single quotes: Example: name = 'Paul'.
- Double quotes: Example: name = "Paul"
- Triple quotes: They are used so that we can work with multi-line strings, and all associated whitespace will be included in the string. Example: """This is a function. Do not change it or else it will not work"""

Use Python's built-in `len()` function to find the length of strings.

You can also index strings. Remember that indexing starts from `0` and not `1`. All the iterables (string, list, dictionaries, tuples, sets) have this property.

Slicing and Indexing in Strings



Indexing

In the above image, for the string `"Monty Python"`, the following points can be noticed:

- From left to right, the index goes from `0` to `length of string minus 1`.
- From right to left, the index goes from `-1` to `-(length of the string)`.
- The first index is at `0`, and it is accessed by `[0]`. It's value is `M`.
- The last index can be accessed by `[-1]` with value `n`.
- Leaving any of the terms before or after `:` means including everything ahead/behind.

Slicing

The syntax for slicing in Python in general is `[Start index (included): Stop index (excluded)]`

In the above example:

- [6:10] returns `Pyth`
- [-12:-7] returns `Monty`

String slicing can also accept a third parameter, `stride`, which refers to how many characters you want to move forward after the first character is retrieved from the string. The value of stride is set to 1 by default.

The actual slicing syntax is `[Start index (included): Stop index (excluded): Stride]`.

Other Common Operations

Let's take the two strings as `a = "Hello"` and `b = "World"`. The following table will be helpful in your journey towards mastering strings in Python:

Operator	Description	Example
`+`	Concatenates two or more strings	a+b $a+b$ gives `"HelloWorld"`
`*`	Repetition: Creates new strings, concatenating multiple copies of the same string	a*2 $a*2$ gives `"HelloHello"`
`[ : ]`	Range Slice: Gives the characters from the given range	a[1:4] $a[1:4]$will give `"ell"`

`in`	Membership: Returns true if a character exists in the given string	`H in a` will give `1`
`not in`	Membership: Returns true if a character does not exist in the given string	`M not in a` will give `1`

Common Built-in String Methods

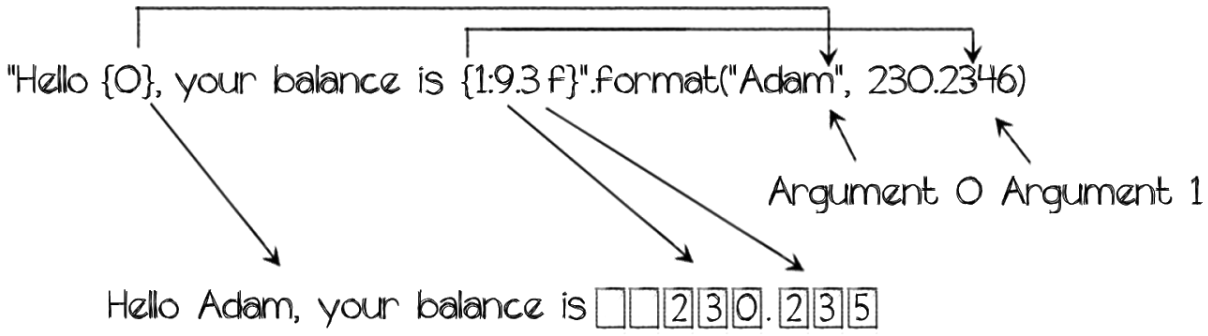
Method	Description
`len()`	Length of the string
`startswith(str, beg=0,end=len(string))`	Determines if a string or a substring of string (if starting index `beg` and ending index `end` are given) starts with substring str; returns true if so, and false otherwise
`endswith(suffix, beg = 0, end = len(a))`	Determines if a string or a substring of a string (if starting index `beg` and ending index `end` are given) ends with suffix; returns `true` if so, and `false` otherwise
`lower()`	Converts all uppercase letters in a string to lowercase
`upper()`	Converts all lowercase letters in a string to uppercase
`split(str="", num=string.count(str))`	Splits a string according to delimiter str (space if not provided) and returns a list of substrings, split into at most num substrings if given

String Formatting

Python's `str.format()` method of the `str` class allows you to carry out variable substitutions and value formatting. Formatters work by putting in one or more replacement fields or placeholders defined by a pair of curly braces {} into a string and calling the str.format() method.

When you pass the value you want to concatenate with the string into the method, this value will appear in the same place that your placeholder is positioned when you run the program.

Let's understand this with the help of an example, shown as follows:



In the above example, we used two formaters to fill in with values "Adam" and "230.2346", rounded to 3 decimal places.

Observe that formaters are denoted by `{}`s and we can put indexes inside it to denote its index. In our example, we put its values inside the formater as `0` and `1`.

Important Property

An important property of strings is that they are immutable, i.e., they cannot be modified once they are created. In order to modify it, you have to create new strings which will have a different memory location in order to do string manipulation. This property is common for immutable data types. In order to check for the memory location of an object, use the `id()` function.

```
s = 'abc'
t = 'abc'

print (id(s) == id(t))
```

Output

```
True
```

For example, let's say you have a `'Soccee'` string, and you want to replace `'e'` with `'r'`. One way to do so is to get the last element and try substituting it with `r`. But we will get an error message saying the `str` object

doesn't support this type of operation. Immutable objects are those that don't allow modifications after creation is inplace, i.e., in the same location in memory.

```
string = 'Soccee'
string[-1] = 'r'
```

Output

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-3-2a65ba8f5e89> in <module>()
      1 string = 'Soccee'
----> 2 string[-1] = 'r'

TypeError: 'str' object does not support item assignment
```

Now, you replace the alphabet `'e'` with `'r'` and can be done by `.replace('a', 'b')`, where we replace 'a' by 'b'. Now, use `id()` to compare the memory locations and as you can see, they are different.

```
# memory loacation of original string
print(id(string))

print('='*20)

# memory location of modified string
print(id(string.replace('e', 'r' )))
```

Output

```
139678449684240
====================
139678448727016
```

How to Create a List?

A List is a collection of elements that are ordered and mutable (i.e., its values can be changed after creation).

Elements of the list are enclosed in square brackets `[]`. You can create a list in Python using square brackets separated by commas and assigning the same to a variable.

Example: `scores = [80, 90, 95, 100]`

Let's say you want to change the value at the second index, i.e., 95 to 99; very simple. Use `scores[2] = 99` and the value changes. But the memory location doesn't change, i.e., it is modified in-place.

```
# initial list
scores = [80,90,95,100]

# memory location of initial list
print('Memory loaction initially is ' , id(scores))
print('='*20)
```

```
# change value
scores[2] = 99
# display modified list
print(scores)
print('='*20)
# check memory loaction of modified list
print('Memory location of modified list is ' , id(scores))
```

Output

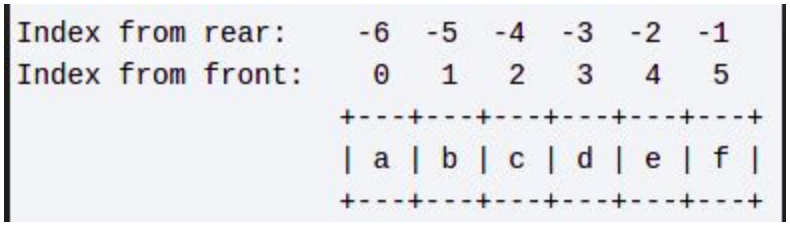
```
Memory loaction initially is  139678449711304
====================
[80, 90, 99, 100]
====================
Memory location of modified list is  139678449711304
```

Indexing

Elements in a list can be accessed using their `index` in the list. `Index` specifies the place of an element in the list. There are two ways to do this which is the same as we saw in strings:

- Forward indexing
- Backward indexing

The following diagram will give you a visual intuition of how a list is in indexed in Python



Indexing and Slicing Lists

The syntax for slicing lists is similar to that of strings:

- `list_object[start:end:step]`

Here, start and end are indices (start inclusive, end exclusive). All slicing values are optional.

Other Common List Operations

There are several operations that can be performed on lists in Python. In this section, we will learn about a few that you would use frequently going forward in this program. Let us take two lists `a = [1,2,3],b = [4,5,6]`.

Operation	Description	Example

`len(a)`	Length	3
`a + b`	Concatenation	`[1,2,3,4,5,6]`
`[a]*2`	Repitition	`[1,2,3,1,2,3]`
`3 in a`	Membership	`True`
`max(a)`	Returns item from the list with max value	3
`min(a)`	Returns item from the list with min value	1
`a.append(10)`	Adds `10` to the end of the list	`[1,2,3,10]`
`a.extend(b)`	Appends the contents of `b` to `a`	`[1,2,3,4,5,6]`
`a.index(1)`	Returns the lowest index in the list that `1` appears in	0
`a.reverse()`	Reverses objects of list in place	`[3,2,1]`
`a.pop()`	Removes and returns last object from list	3
`a.remove(2)`	Removes object `2` from list	`[1,3]`

What are Dictionaries?

Dictionaries are similar to hash or maps in other languages. It consists of key-value pairs where value can be accessed by unique key in the dictionary.. But what's the difference between lists and dictionaries?

Unique Features

- Lists are ordered sets of objects, whereas dictionaries are unordered sets.
- Items in dictionaries are accessed via keys and not via their position.
- Since values of a dictionary can be any Python data type, so dictionaries are unordered key-value-pairs.

- Dictionaries don't support the sequence operation of the sequence data types like strings, tuples and lists.
- They belong to the built-in mapping type. They are the sole representative of this kind!

Creating a Dictionary

```
>>> myDictionary = {"key1": "value1", "key2":"value2", "key3": "value3"}
>>> keys = myDictionary.keys()
>>>
>>> print(keys)
dict_keys(['key2', 'key3', 'key1'])
>>>
>>> myDictionary["newKey"] = "newValue"
>>>
>>> print(keys)
dict_keys(['key2', 'key3', 'newKey', 'key1'])
```

In the figure above,

- A dictionary is enclosed in `{}`.
- Every element in the dictionary has two components namely the `key`, and `value`. In the above example "`key1`", "`key2`", "`key3`" are the keys and "`value1`", "`value2`", and "`value3`" are their respective values.
- We can access the values by making use of these keys; for example, typing `myDictionary["key1"]` will give us "`value1`".
- We can have different data types for different values in a dictionary.
- Use the `.keys()` method to look at all the keys as shown.
- To add a new key-value pair, just look at the third line in the image. A new key named "`newKey`" and its value "`newValue`" is created by `myDictionary["newKey"] = "newValue"`.

Dictionary keys are mutable (can't be listed), and all the keys are unique (no duplicates allowed).

Dictionary Operations

To understand dictionary operations we will use the following dictionary:

`area = { 'living' : [400, 450], 'bedroom' : [650, 800], 'kitchen' : [300, 250], 'garage' : [250, 0]}.`

- Accessing Values in a Dictionary
  We can access the values within a dictionary using keys and perform various operations on it.
  Example: If you need to access areas of bedrooms, and you have already have seen that `bedroom` is a key in the dictionary `area`, you can access it using `area['bedroom']`.
  Another method by which you can accomplish the same thing is `dict.get(key, default_value)`. Here, you will search for `key` in the keys of `dict` and, if present, it returns the value associated with that key. And if the `key` is absent, then the `default_value` gets displayed. Show below is an example carried out on `area`:

```
area = { 'living' : [400,450], 'bedroom' : [650, 800], 'kitchen' : [300, 250], 'garage': [250,
0]}
print(area.get('living'))
print('='*50)
print (area.get('color','This key is absent' ))
```

Output

```
[400, 450]
==================================================
This key is absent
```

- Updating Values in a Dictionary
  We can update a dictionary by adding a new entry or a key-value pair or modifying an existing entry
  Example:
  - To update the value of `living` to `[400,500]`, we can simply use `area['living'] = [400, 500]`
  - Suppose now we also have area of `garden`, which needs to be updated in the dictionary, we can use `area['garden'] = [200, 250]`.
- Deleting Values in a Dictionary
  We can delete values for a particular `key` of a dictionary using the `del` command
  Example: To delete the key-value pair for `'garage'`, we can use `del area['garage']`
- Updating a Dictionary
  Now you want the dictionary `area` to have another key `color` and the `['red', 'blue']` list. This process is called updating the dictionary and can be achieved by `dict1.update(dict2)` where `dict1` and `dict2` are two dictionaries.

```
area = { 'living' : [400,450], 'bedroom' : [650, 800], 'kitchen' : [300, 250], 'garage': [250,
0]}
d2 = {'color' :['red','blue']}
area.update(d2)
print(area)
```

Output

```
{'living': [400, 450], 'bedroom': [650, 800], 'kitchen': [300, 250], 'garage': [250, 0],
'color': ['red', 'blue']}
```

Creating a Tuple

Tuples are another type similar to lists. They store values of any type separated by commas. But how do they differ from lists?

- Firstly, they start and end with parenthesis `()`. For example, (1,2,3,4).
- But the main difference that unlike lists, tuples are immutable i.e. they cannot be updated/changed. You can think of them as read-only lists.

As discussed above, it should begin and end with parenthesis.

Taking the same example of weights from our discussion on `lists`,

```
scores = (80,90,95,100)
```

The only difference is that now we cannot change the values inside it.

Common Tuple Operations

Let us take two tuples `a = (1,2)` and `b = (3,4)` for this purpose.

Expression	Description	Example
`len(a)`	Length of tuple	2
`a + b`	Concatenation	`(1,2,3,4)`
`a*4`	Repitition	`(1,2,1,2,1,2,1,2)`
`2 in a`	Membership	`True`
`max(a)`	Maximum value in tuple	2
`min(a)`	Minimum value in tuple	1
`a[0]`	Indexing	1
`a[:2]`	Slicing	`(1,2)`

You cannot delete an element from the tuple (as they are immutable), but you can delete the entire tuple.

Introduction

You have been hearing about mutable and immutable all along. Simply put, objects which can be modified after creation in the same memory location are called mutable and the others which cannot are termed immutable.

Immutable Objects: int, float, long, complex, string, tuple, bool

Mutable Objects: list, dict, set, byte array, user-defined classes

Checking Mutability

You can check if an object is mutable by first modifying the object and then comparing its new memory location with the old memory location. You can check for memory location either by using the `id()` function, which gives the memory location of an object or with the help of `is` operator, which checks for the identity of two objects.

First, let's take an integer (type `int`) 50 and add 1 to it with the same variable name, then check its memory location before and after modification with `id()`. As it turns out, both have different memory locations, and hence it is an immutable type.

```
#initial variable

a = 50

# initial memory location

print(id(a))

#modified variable

a += 1

# new memory location, is it the same?

print(id(a))
```

Output

```
94285850046496

94285850046528
```

Now, let's take a list with values `[1,2,3,4]` and add 5 to it with `.append()`. You observe that after modification, it still refers to the same memory location as before and hence it is of mutable type.

```
# intial list

l = [1,2,3,4]

# initial memory location

print(id(l))

print('='*20)

# new list

l.append(5)

print(l)

print('='*20)

# new memory location
```

```
print(id(l))
```

Output

```
139678448773256

====================

[1, 2, 3, 4, 5]

====================

139678448773256
```

Introduction

Arithmetic operators include signs like `+, -, *, /` etc. These signs help us carry out simple mathematical operations like addition, subtraction, and multiplication.

Operator	Description	Example
+(Addition)	Adds values on either side of the operator	2 + 4
- (Subtraction)	Subtracts the right-hand operand from left-hand operand	4 - 2
* (Multiplication)	Multiplies values on either side of the operator	2 * 4
/ (Division)	Divides the left hand operand by the right hand operand	4 / 2 = 2.0
% (Modulus)	Divides the left-hand operand by the right hand operand and returns remainder	4 % 2 = 0
** (Exponent)	Exponential (power) calculation on operators	4^2 4 2

// (Floor Division)	Division of operands where the result is the quotient and the digits after the decimal point are removed. But if one of the operands is negative, the result is floored, i.e., rounded away from zero (towards negative infinity):	`9//2 = 4`, `9.0//2.0 = 4.0`, `-11//3 = -4`, `-11.0//3 = -4.0`

Introduction

Also called Relational operators, they compare the values on either side of the operand and decide the relation among them.

Operator	Description	Example
==	If the values of the two operands are equal, then the condition becomes true	(4 == 4) is true
!=	If values of the two operands are not equal, then the condition becomes true	(2!= 4) is true
>	If the value of the left operand is less than the value of the right operand, then the condition becomes false	(2 > 4) is false
<	If the value of the left operand is less than the value of the right operand, then the condition becomes true	(2 < 4) is true
>=	If the value of the left operand is less than or equal to the value of the right operand, then the condition becomes false	(2 >= 4) is false
<=	If the value of the left operand is less than or equal to the value of the right operand, then the condition becomes true	(2 <= 4) is true

Logical operators are the `and`, `or`, `not` operators.

Operator	Description	Example
`and` (Logical AND)	If both the operands are true, then condition becomes true	(*True* and *False*) is False
`or` (Logical OR)	If any of the two operands are non-zero then condition becomes true	(*True* or *False*) is True
`not` (Logical NOT)	Used to reverse the logical state of its operand	not(*False*) is True

Assignment Operators

Operator	Description	Example
`=`	Assigns values from the right side operands to the left side operands	`x = a + b` assigns value of a + b into x
`+=` (Add AND)	Adds the right operand to the left operand and assigns the result to the left operand	`x += a` is equivalent to `x = x + a`
`-=` (Subtract AND)	Subtracts the right operand from the left operand and assigns the result to the left operand	`x -= a` is equivalent to `x = x - a`
`*=` (Multiply AND)	Multiplies the right operand with the left operand and assigns the result to the left operand	`x *= a` is equivalent to `x = x*a`
`/=` (Divide AND)	Divides the left operand with the right operand and assigns the result to the left operand	`x /= a` is equivalent to `x = x / a`
`%=` (Modulus AND)	Takes the modulus using two operands and assigns the result to left operand	`x %= a` is equivalent to `x = x % a`
`**=` (Exponent AND)	Performs exponential (power) calculation on operators and assigns value to the left operand	`x **= a` is equivalent to `x = x ** a`

`//=` (Floor Division)	Performs the floor division on operators and assigns value to the left operand	`x //= a` is equivalent to `x = x // a`

Bitwise Operators

Bitwise operators act on operands as if they were a string of binary digits. It operates bit by bit, hence the name.

For example, `2` is `10` in binary and `7` is `111`.

In the table below: Let `x = 10` (`0000 1010` in binary) and `y = 4` (`0000 0100` in binary)

Operator	Description	Example
&	Bitwise AND	$x\&y = 0(00000000)$
|	Bitwise OR	$x\|y = 14(00001110)$
~	Bitwise NOT	$\sim x = -11(11110101)$
^	Bitwise XOR	$x^y = 14(00001110)$
>>	Bitwise right shift	$x >> 2 = 2(00000010)$
<<	Bitwise left shift	$x << 2 = 40(00101000)$

Introduction

Membership operators test for membership in a sequence (string, list, tuple, set and dictionary). You will learn about strings, sets, tuple, etc. in the upcoming chapters.

Operator	Description	Example
`in`	Evaluates to true if it finds a variable in the specified sequence and false otherwise	`x` in `y`, herein results in a 1 if `x` is a member of sequence `y`
`not in`	Evaluates to true if it does not finds a variable in the specified sequence and false otherwise	`x` not in `y`, here not in results in a 1 if `x` is not a member of sequence `y`

Introduction

Identity operators compare the memory locations of two objects. You already know by now that we use the `id()` to check the memory location of an object. There are two Identity operators, as explained below:

Operator	Description	Example
`is`	Evaluates to true if the variables on either side of the operator point to the same object and false otherwise	`x` is `y`, here is results in `1` if `id(x)` equals `id(y)`
`is not`	Evaluates to false if the variables on either side of the operator point to the same object and true otherwise	`x` is not `y`, here is not results in `1` if `id(x)` is not equal to `id(y)`