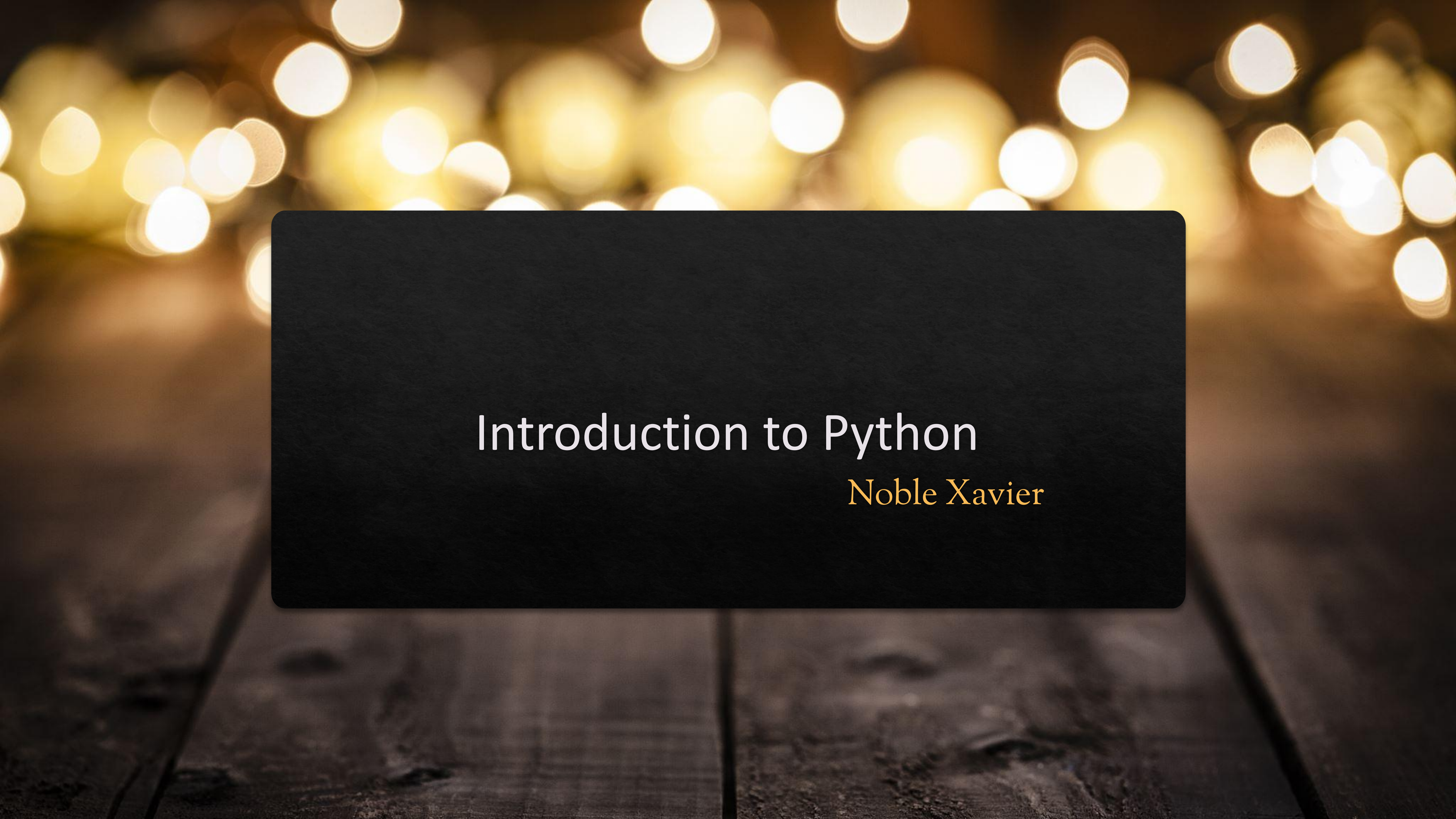
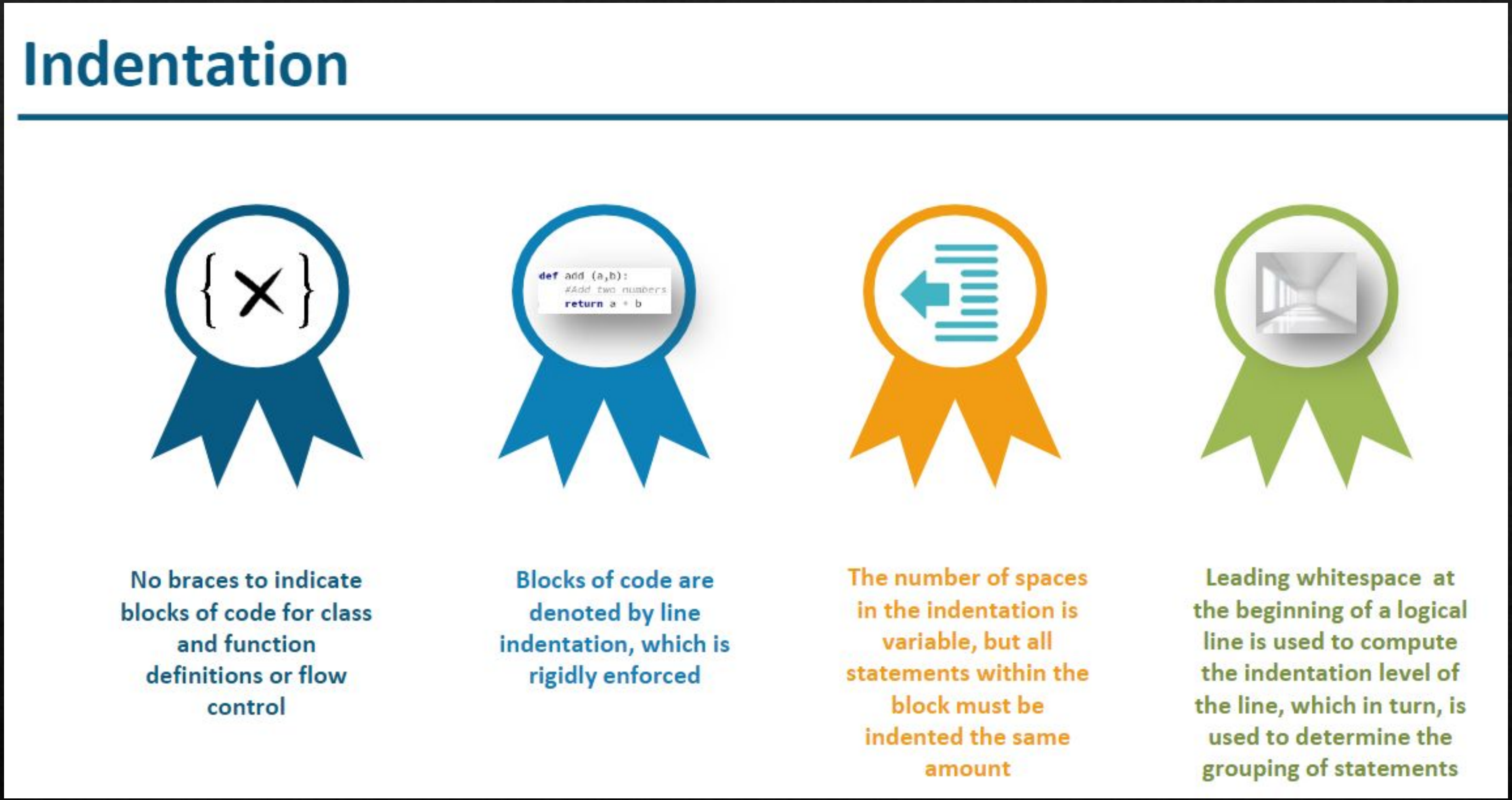


Introduction to Python

Noble Xavier



Indentation

```
def add (a,b):
    #Add two numbers
    return a + b
```

No braces to indicate blocks of code for class and function definitions or flow control

Blocks of code are denoted by line indentation, which is rigidly enforced

The number of spaces in the indentation is variable, but all statements within the block must be indented the same amount

Leading whitespace at the beginning of a logical line is used to compute the indentation level of the line, which in turn, is used to determine the grouping of statements

Identifiers – Naming Conventions

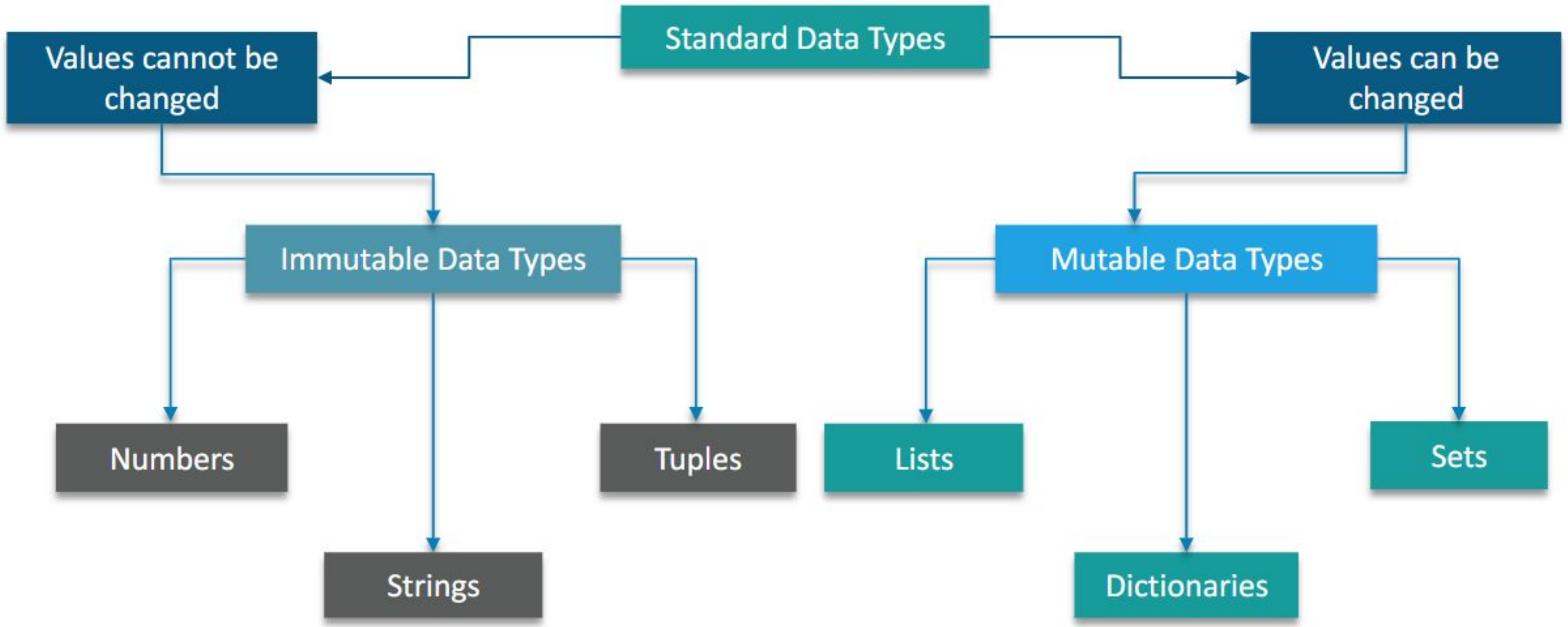
Class names start with an uppercase letter. All other identifiers start with a lowercase letter

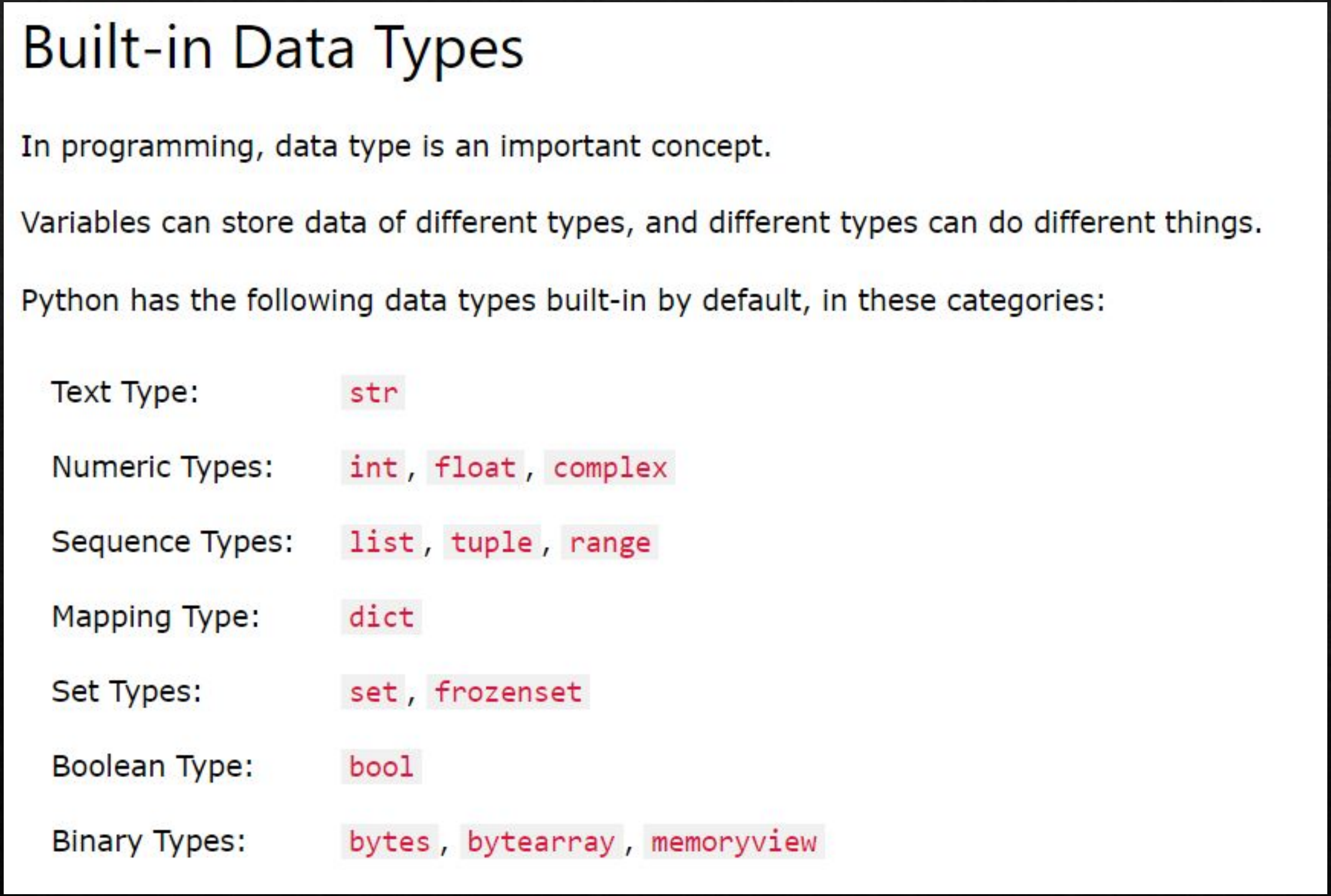
Starting an identifier with a single leading underscore indicates that the identifier is private

Starting an identifier with two leading underscores indicates a strongly private identifier

If the identifier also ends with two trailing underscores, the identifier is a language-defined special name

Standard Data Types





Built-in Data Types

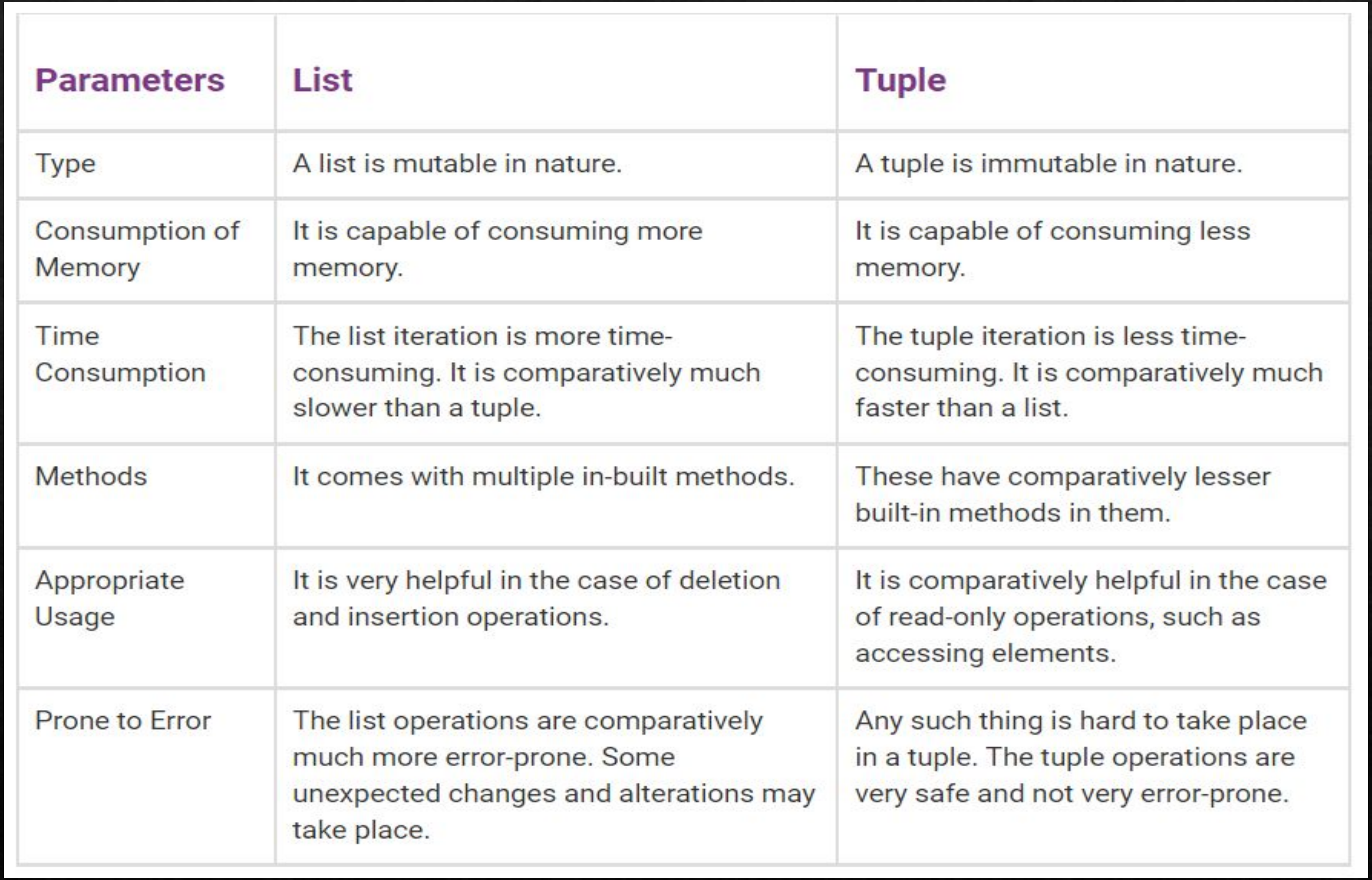
In programming, data type is an important concept.

Variables can store data of different types, and different types can do different things.

Python has the following data types built-in by default, in these categories:

Text Type:	`str`
Numeric Types:	`int`, `float`, `complex`
Sequence Types:	`list`, `tuple`, `range`
Mapping Type:	`dict`
Set Types:	`set`, `frozenset`
Boolean Type:	`bool`
Binary Types:	`bytes`, `bytearray`, `memoryview`

Difference Between List and Tuple in Python



Parameters	List	Tuple
Type	A list is mutable in nature.	A tuple is immutable in nature.
Consumption of Memory	It is capable of consuming more memory.	It is capable of consuming less memory.
Time Consumption	The list iteration is more time-consuming. It is comparatively much slower than a tuple.	The tuple iteration is less time-consuming. It is comparatively much faster than a list.
Methods	It comes with multiple in-built methods.	These have comparatively lesser built-in methods in them.
Appropriate Usage	It is very helpful in the case of deletion and insertion operations.	It is comparatively helpful in the case of read-only operations, such as accessing elements.
Prone to Error	The list operations are comparatively much more error-prone. Some unexpected changes and alterations may take place.	Any such thing is hard to take place in a tuple. The tuple operations are very safe and not very error-prone.

Difference Between List and Tuple in Python

Operations

Although there are many operations similar to both lists and tuples, lists have additional functionalities that are not available with tuples. These are insert and pop operations, and sorting and removing elements in the list.

Size

In Python, tuples are allocated large blocks of memory with lower overhead, since they are immutable; whereas for lists, small memory blocks are allocated. Between the two, tuples have smaller memory. This helps in making tuples faster than lists when there are a large number of elements.

Difference Between List and Tuple in Python

Length

Lengths differ in the two data structures. Tuples have a fixed length, while lists have variable lengths. Thus, the size of created lists can be changed, but that is not the case for tuples.

Debugging

When it comes to debugging, in lists vs tuples, tuples are easier to debug for large projects due to its immutability. So, if there is a smaller project or a lesser amount of data, it is better to use lists. This is because lists can be changed, while tuples cannot, making tuples easier to track.

bytes, bytearray, memoryview

It can store Values from 0 to 255

bytes and bytearray are used for manipulating binary data. The memoryview uses the buffer protocol to access the memory of other binary objects without needing to make a copy.

Bytes objects are immutable sequences of single bytes. We should use them only when working with ASCII compatible data.

The syntax for bytes literals is same as string literals, except that a 'b' prefix is added.

bytearray objects are always created by calling the constructor bytearray(). These are mutable objects.

bytes, bytearray, memoryview

Bytes

- It can store Values from 0 to 255

example -1

```
b1=  bytes ([10,20,30,255])
print (b1) # Print Bytes
for i in b1:
    print (i)
```

example -2- Error – since value more than 255

```
b1=  bytes ([10,20,30,300])
print (b1) # Print Bytes
```

bytes, bytearray, memoryview

Bytes

- It is immutable - Item assignment not possible

example -1 – This is not possible since immutable

```
b1[1] =25
```

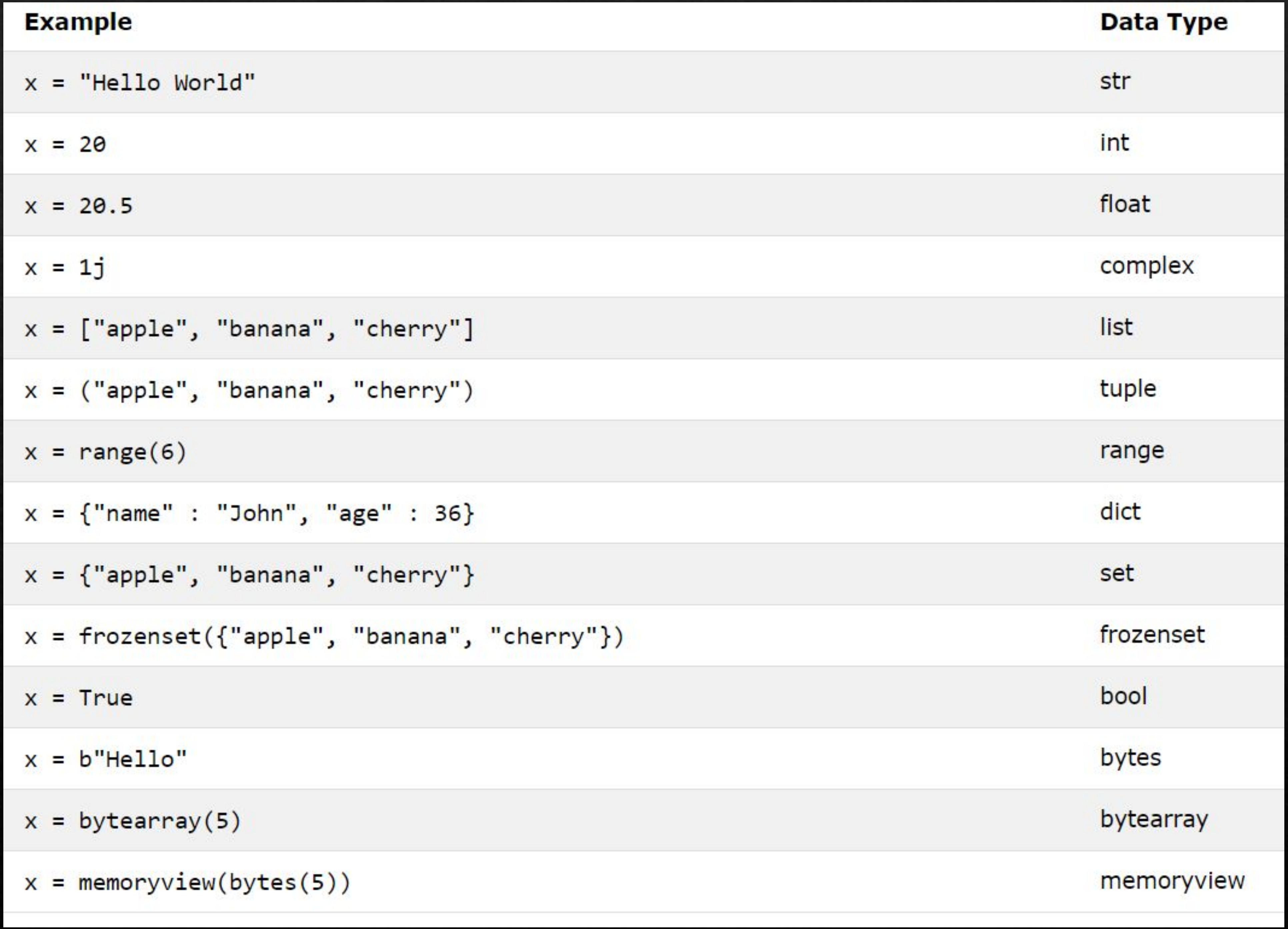
bytes, bytearray, memoryview

Bytearray

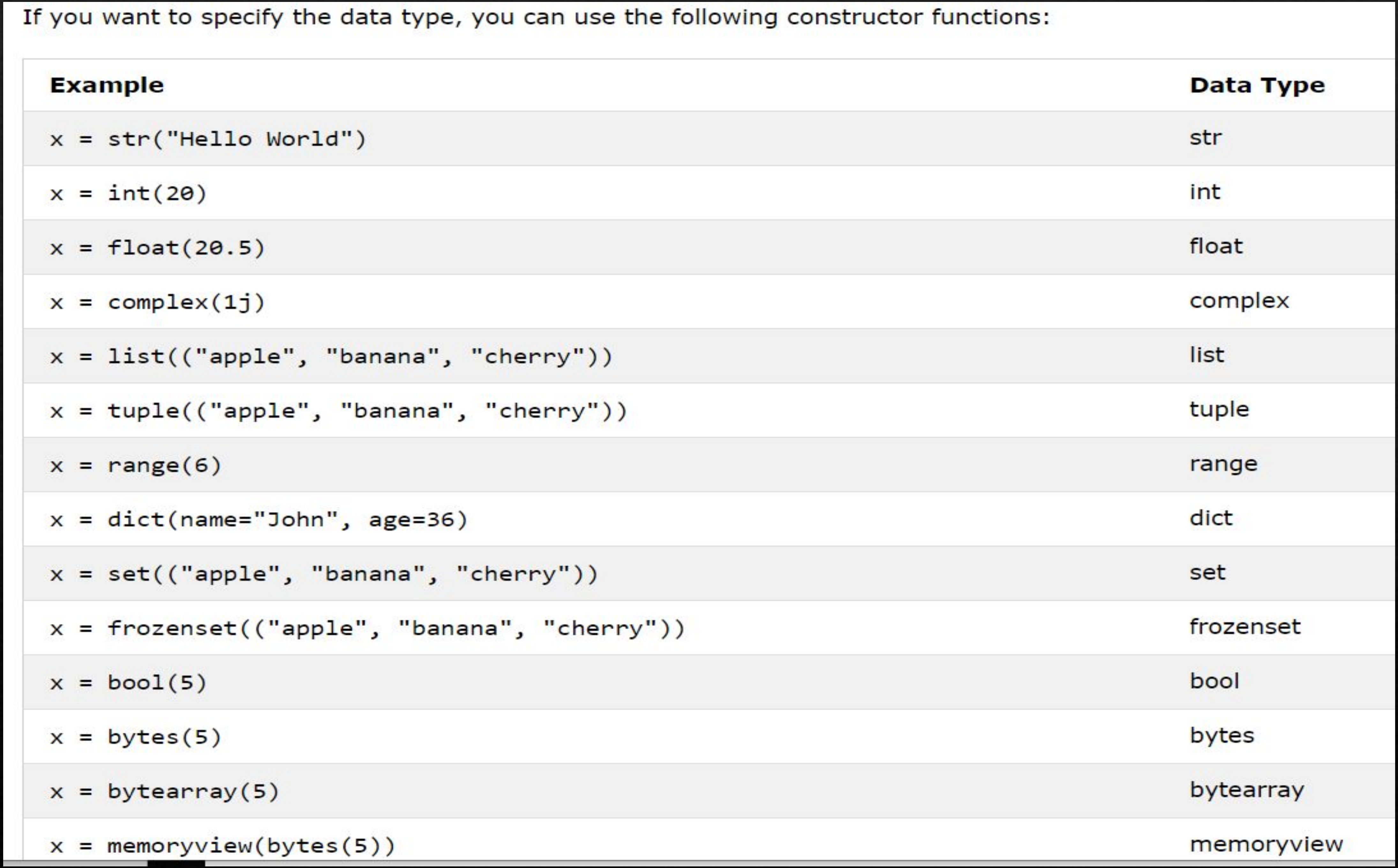
- It is mutable - Item assignment possible

example -1

```
b1= bytearray([10,20,30,255])
print (b1)
b1[1]=100 # update first element with 100
for i in b1:
    print (i)
```

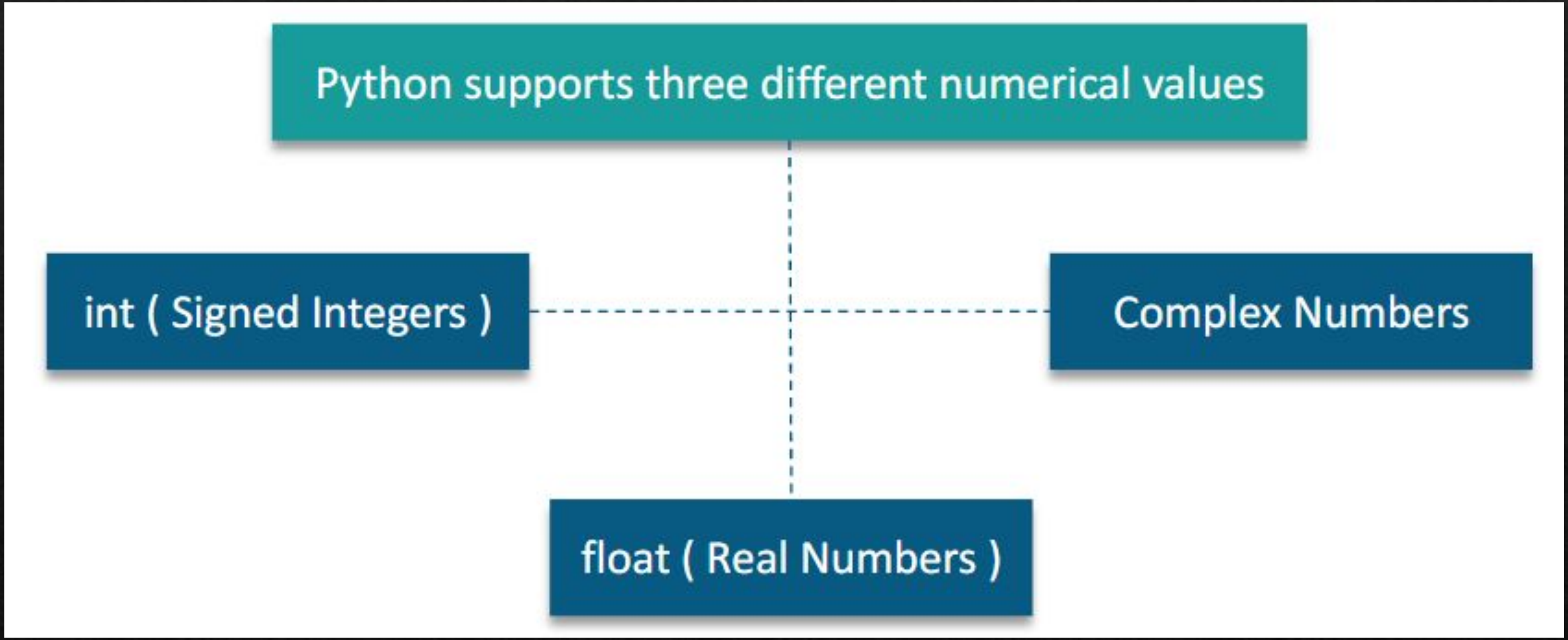


Example	Data Type
`x = "Hello World"`	str
`x = 20`	int
`x = 20.5`	float
`x = 1j`	complex
`x = ["apple", "banana", "cherry"]`	list
`x = ("apple", "banana", "cherry")`	tuple
`x = range(6)`	range
`x = {"name" : "John", "age" : 36}`	dict
`x = {"apple", "banana", "cherry"}`	set
`x = frozenset({"apple", "banana", "cherry"})`	frozenset
`x = True`	bool
`x = b"Hello"`	bytes
`x = bytearray(5)`	bytearray
`x = memoryview(bytes(5))`	memoryview



If you want to specify the data type, you can use the following constructor functions:

Example	Data Type
`x = str("Hello World")`	str
`x = int(20)`	int
`x = float(20.5)`	float
`x = complex(1j)`	complex
`x = list(("apple", "banana", "cherry"))`	list
`x = tuple(("apple", "banana", "cherry"))`	tuple
`x = range(6)`	range
`x = dict(name="John", age=36)`	dict
`x = set(("apple", "banana", "cherry"))`	set
`x = frozenset(("apple", "banana", "cherry"))`	frozenset
`x = bool(5)`	bool
`x = bytes(5)`	bytes
`x = bytearray(5)`	bytearray
`x = memoryview(bytes(5))`	memoryview


Python supports three different numerical values
int (Signed Integers)
Complex Numbers
float (Real Numbers)

Sets - Mutable Data Type

Lists

Dictionaries

Sets

A set is an unordered collection of items. Every element is unique. A set is created by placing all the items (elements) inside curly braces {}, separated by comma.

```
A={1,2,3,3}

print(A)
```

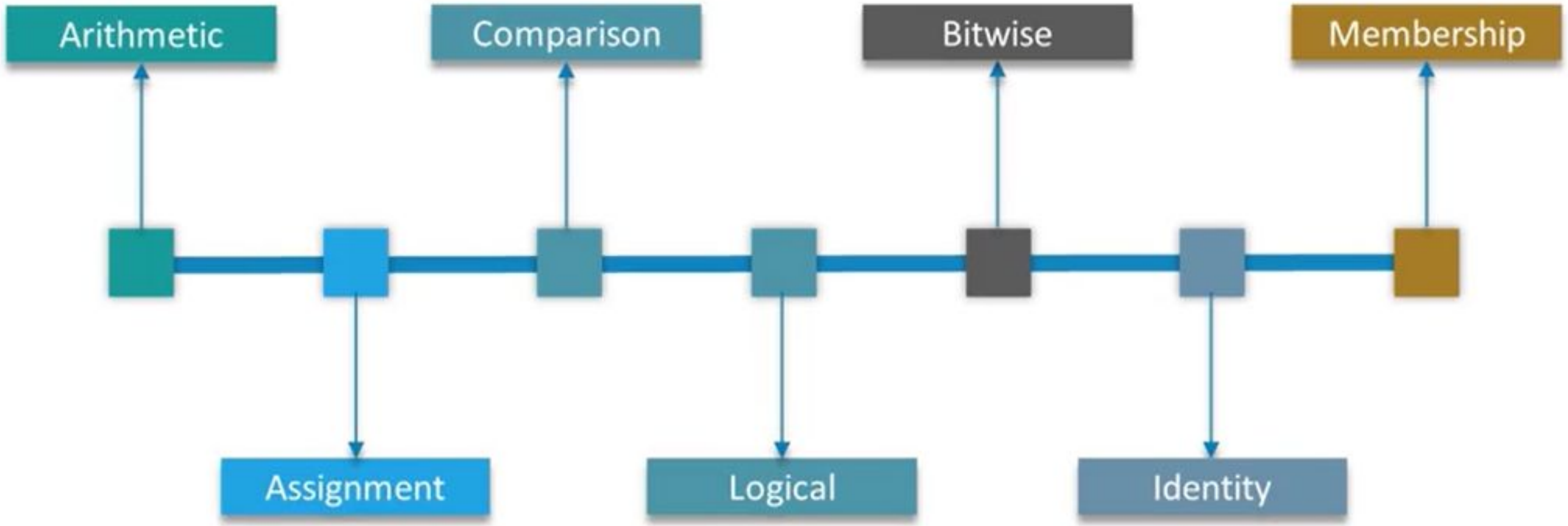
Every element in a Set has to be unique

```
{1, 2, 3}
```

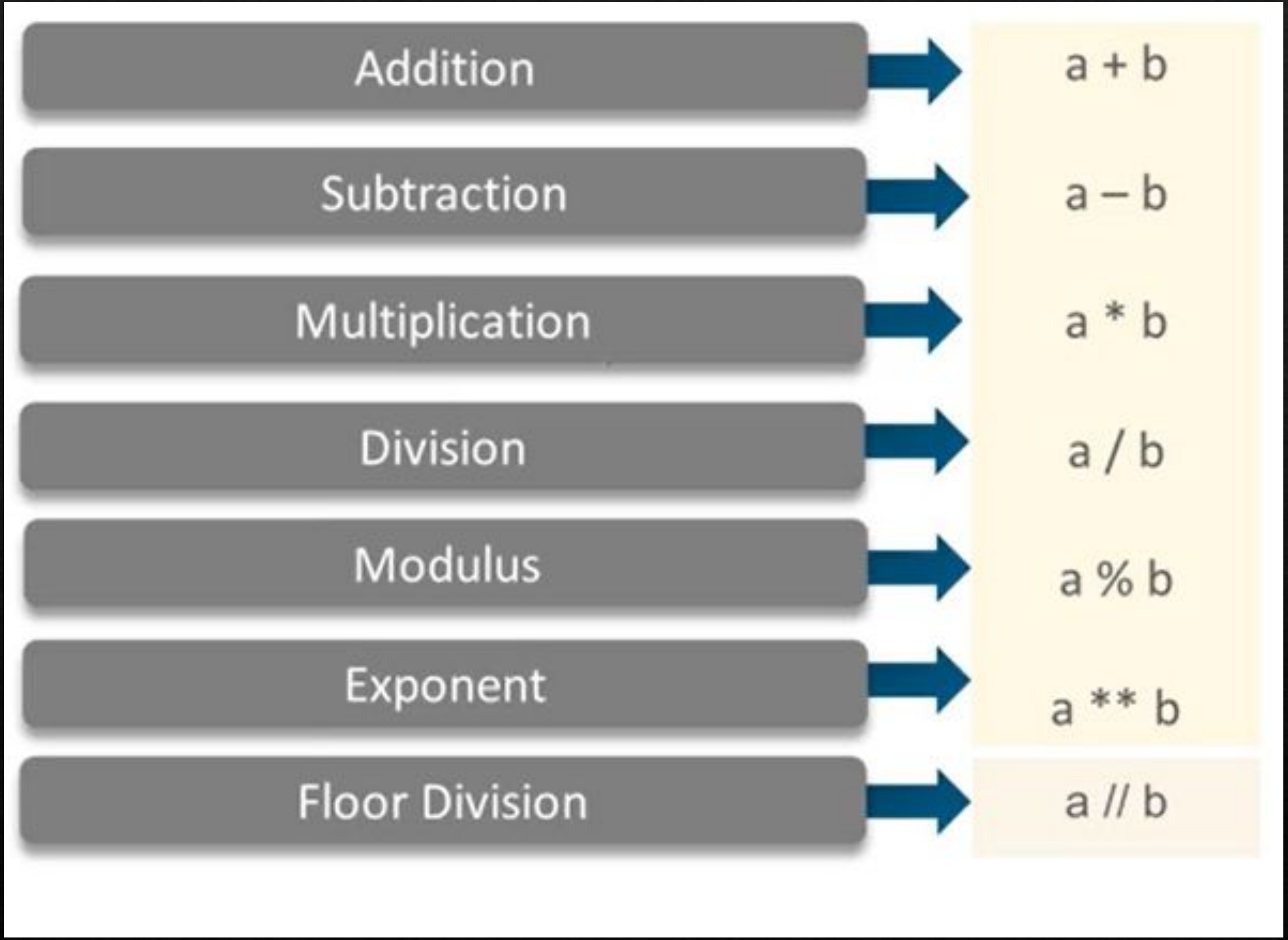
Output – Notice that 3 has appeared only once

Operators

Operators are the constructs which can manipulate the values of the Operands. Consider the expression 2 + 3 = 5, here 2 and 3 are Operands and + is called Operator

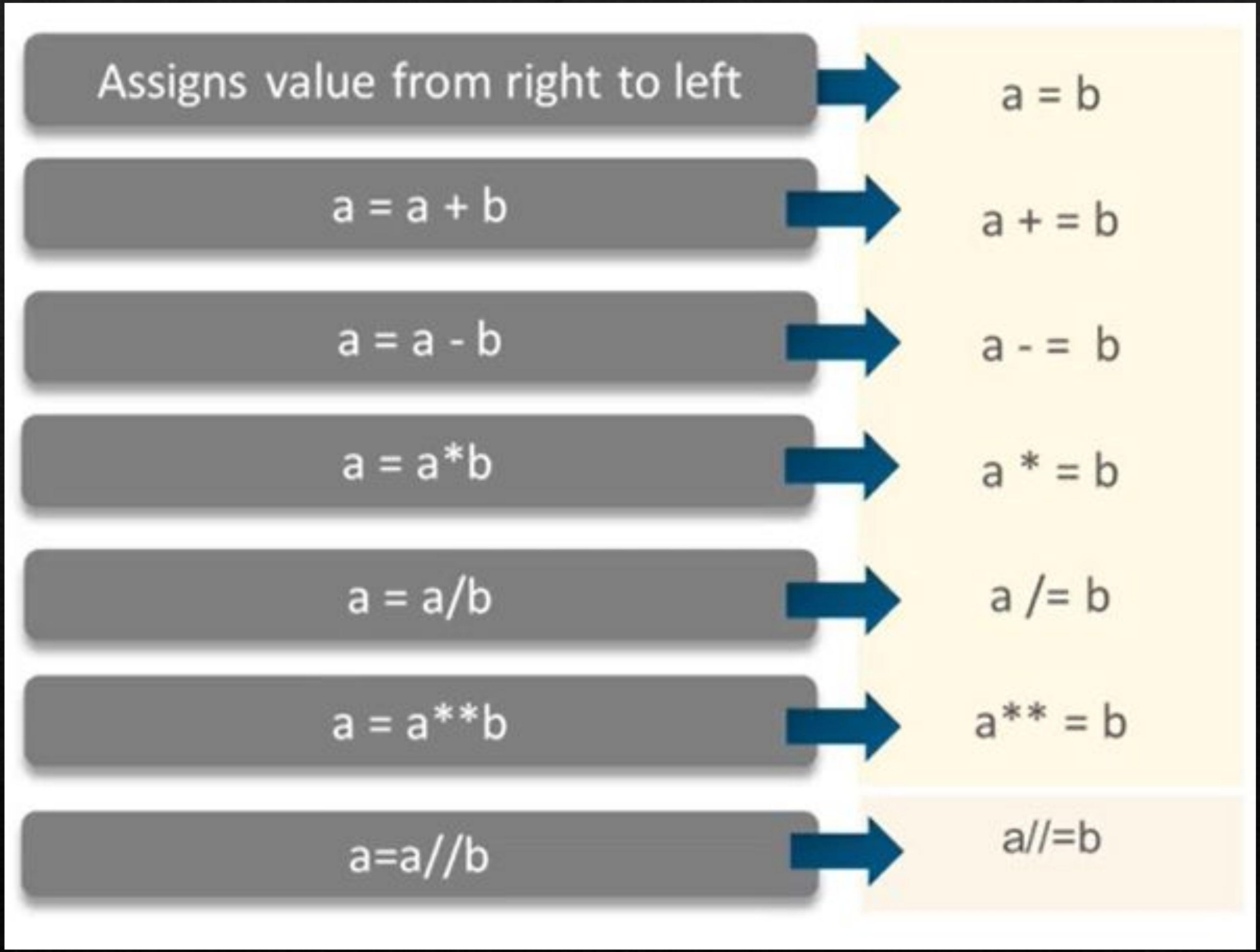


Arithmetic Operator

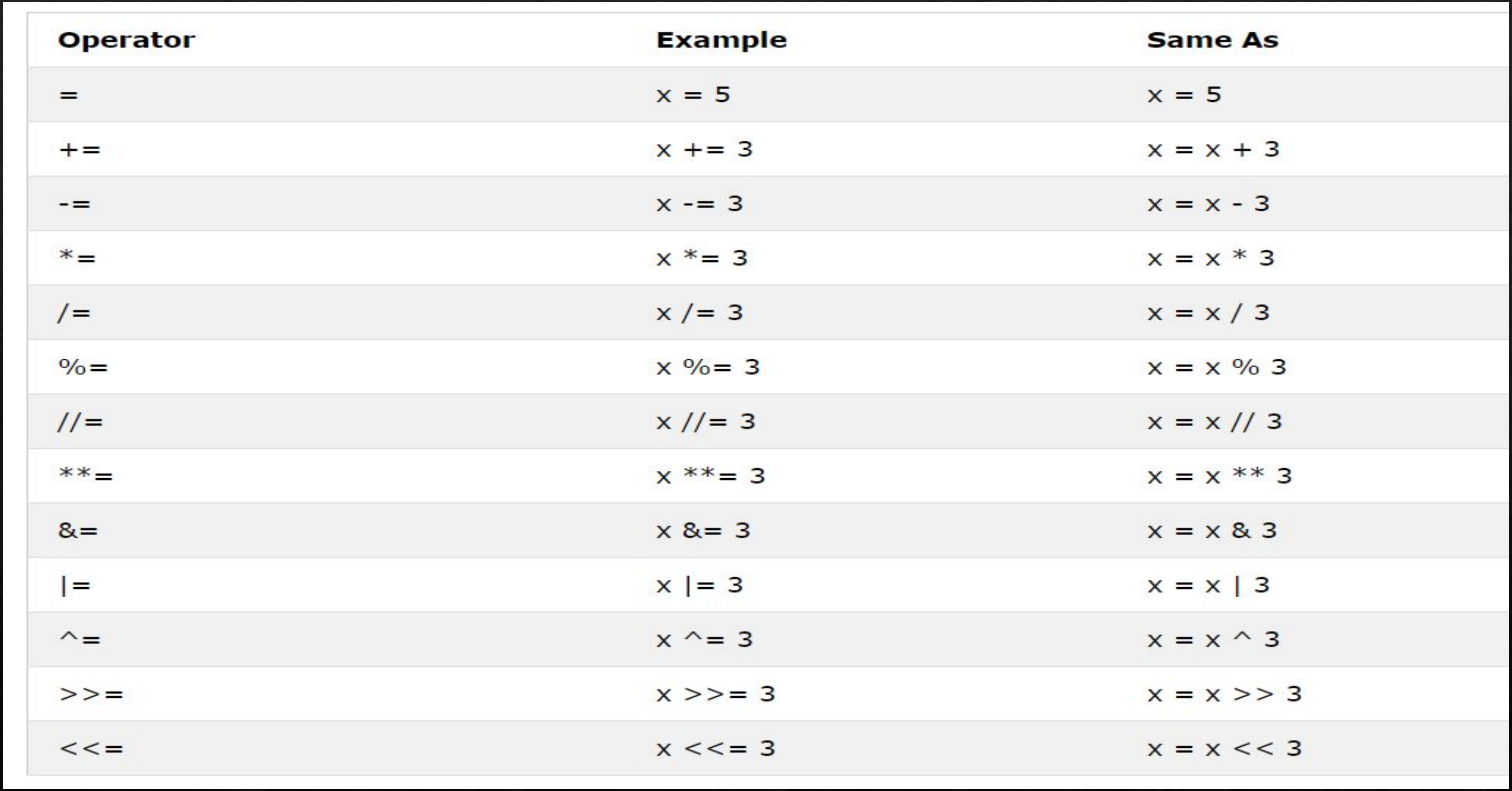


Addition	a + b
Subtraction	a – b
Multiplication	a * b
Division	a / b
Modulus	a % b
Exponent	a ** b
Floor Division	a // b

Assignment Operator

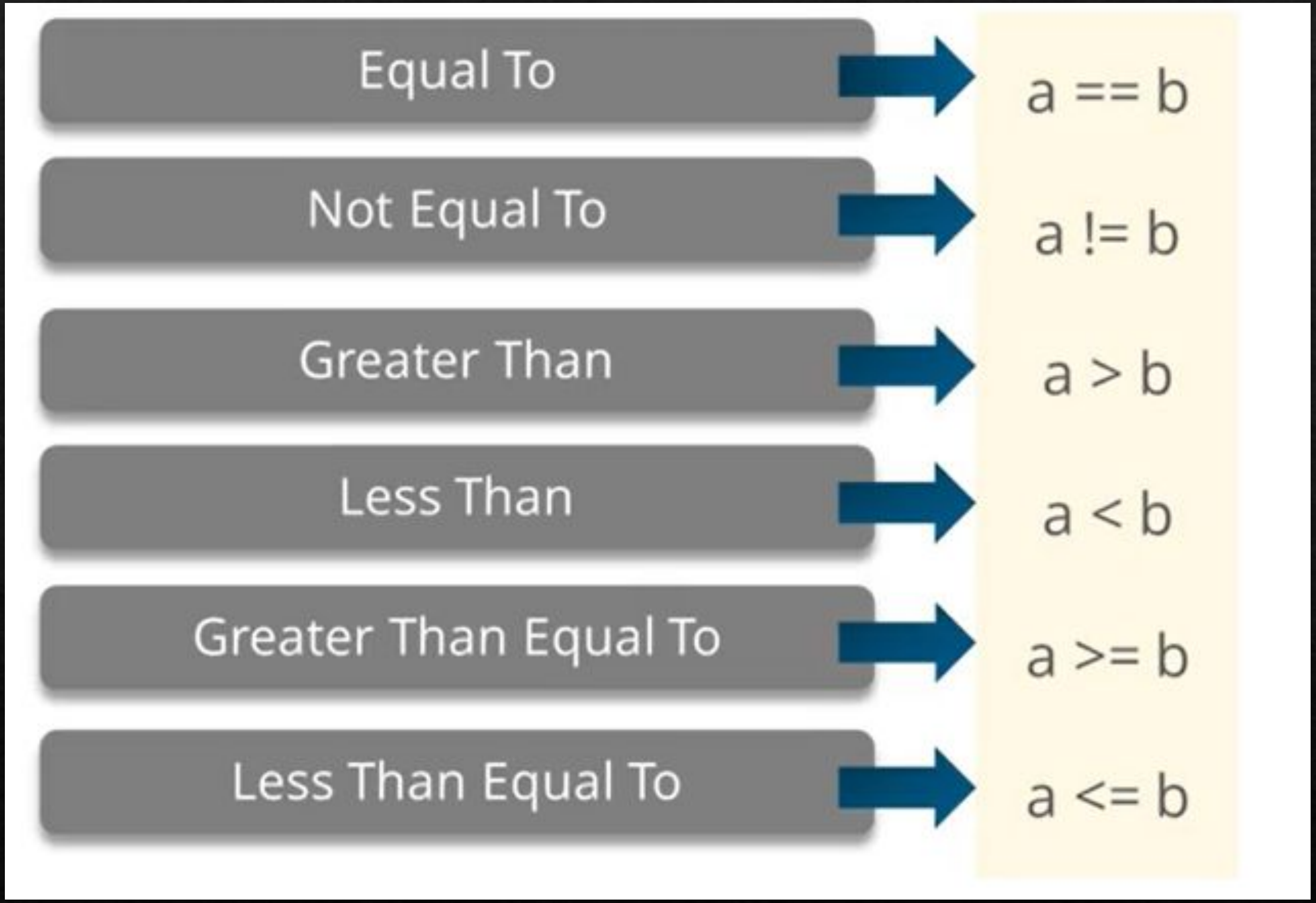


Assigns value from right to left	a = b
a = a + b	a + = b
a = a - b	a - = b
a = a*b	a * = b
a = a/b	a /= b
a = a**b	a** = b
a=a//b	a//=b



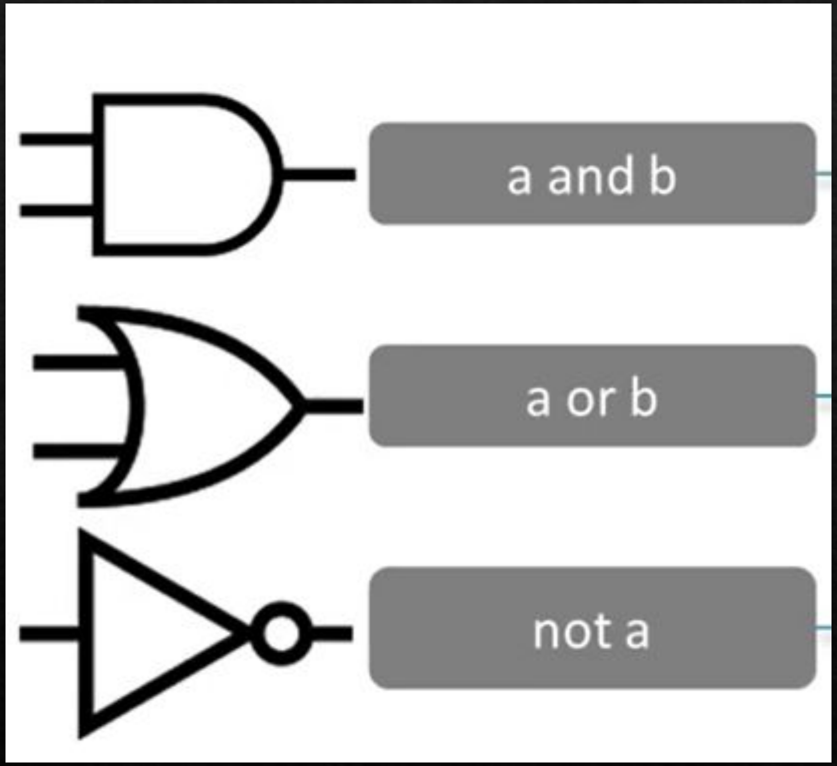
Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
//=	x //= 3	x = x // 3
**=	x **= 3	x = x ** 3
&=	x &= 3	x = x & 3
|=	x |= 3	x = x | 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

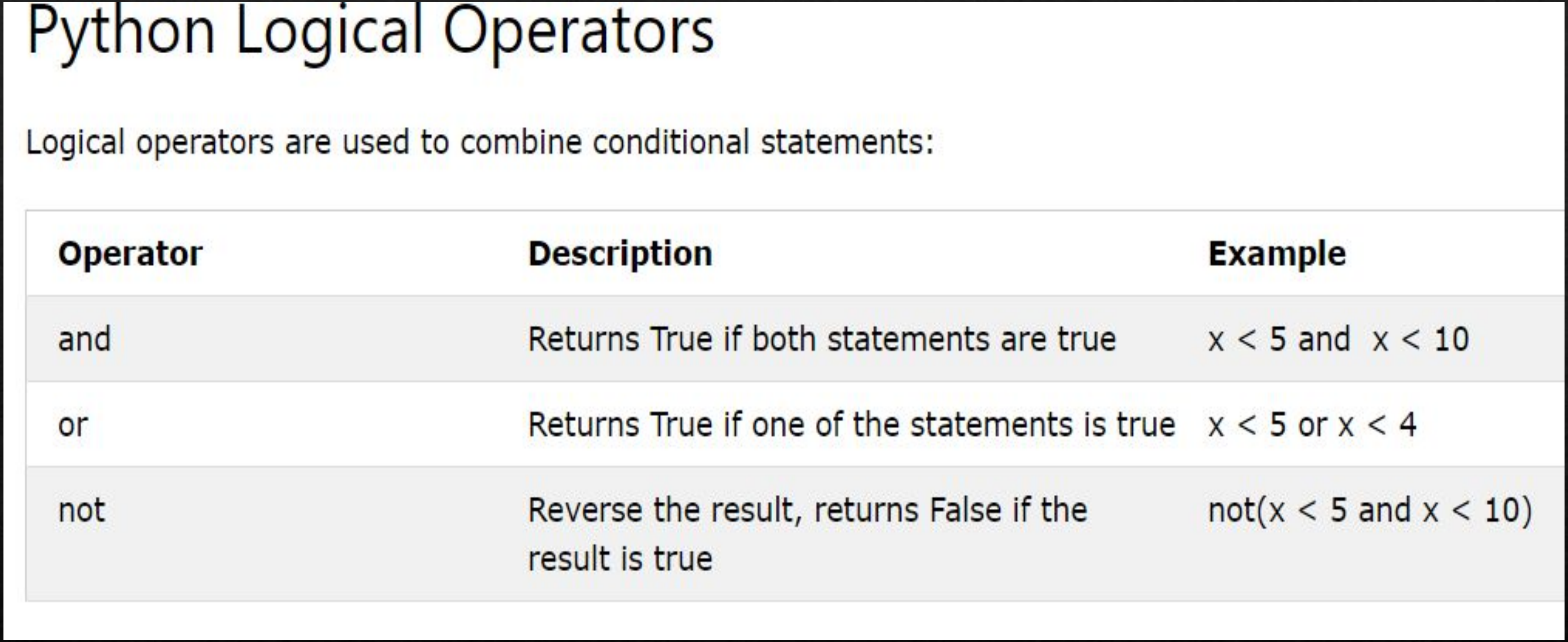
Comparison True / False



Equal To	a == b
Not Equal To	a != b
Greater Than	a > b
Less Than	a < b
Greater Than Equal To	a >= b
Less Than Equal To	a <= b

Logical Operator



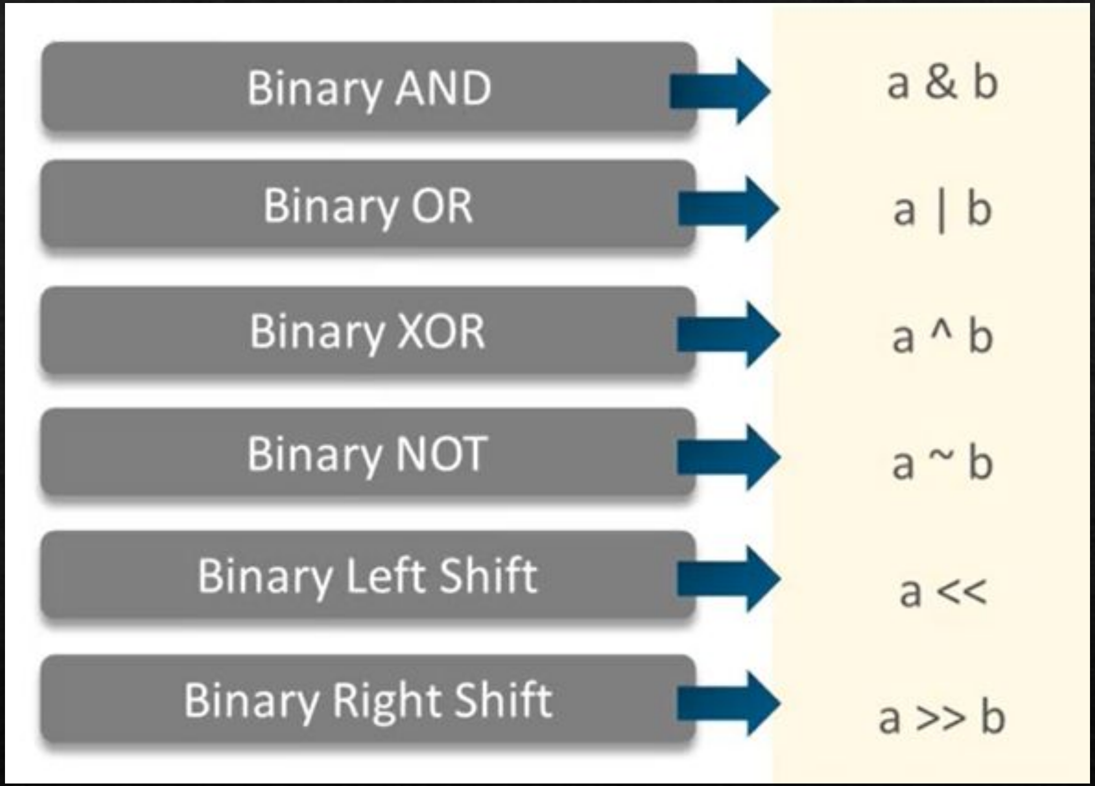


Python Logical Operators

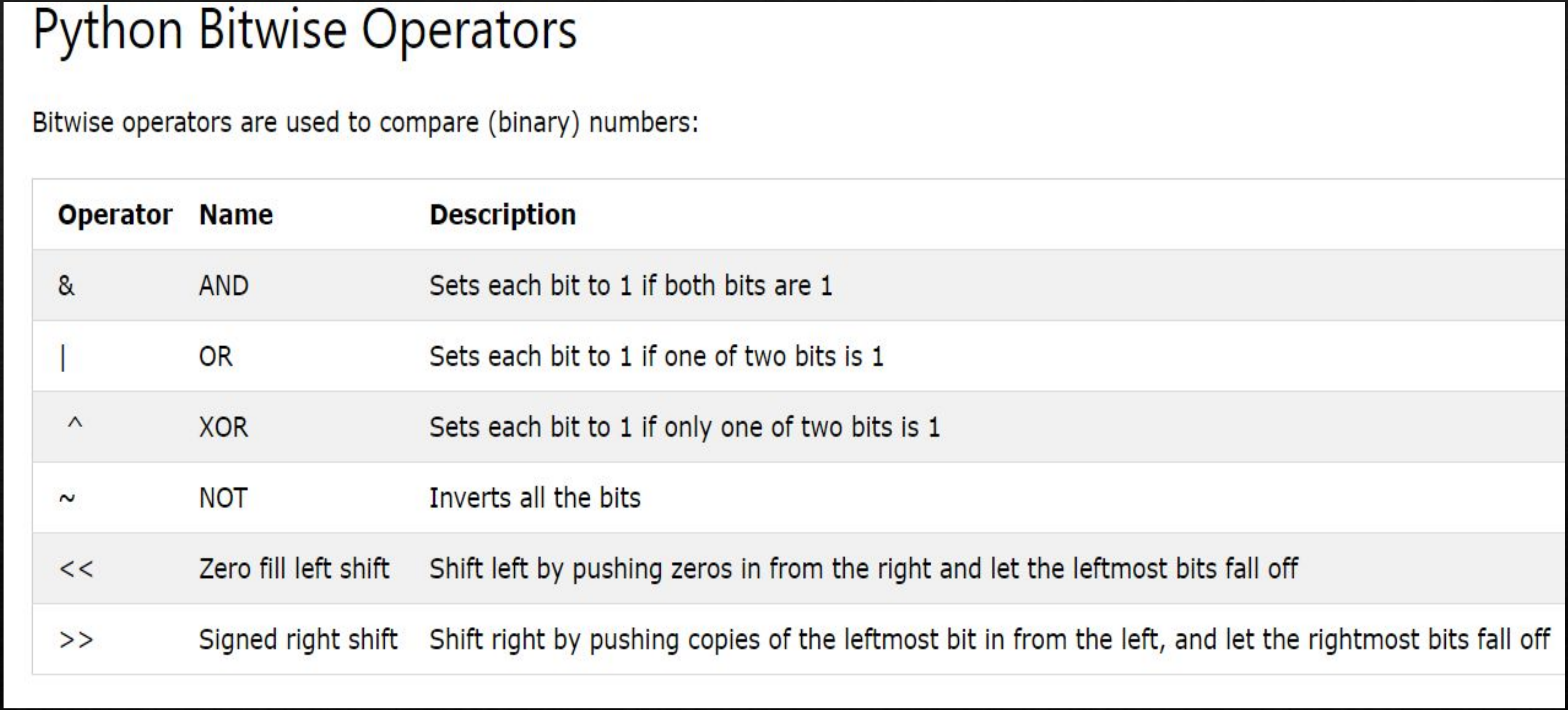
Logical operators are used to combine conditional statements:

Operator	Description	Example
and	Returns True if both statements are true	x < 5 and x < 10
or	Returns True if one of the statements is true	x < 5 or x < 4
not	Reverse the result, returns False if the result is true	not(x < 5 and x < 10)

Bitwise -Operator



Binary AND	a & b
Binary OR	a | b
Binary XOR	a ^ b
Binary NOT	a ~ b
Binary Left Shift	a <<
Binary Right Shift	a >> b

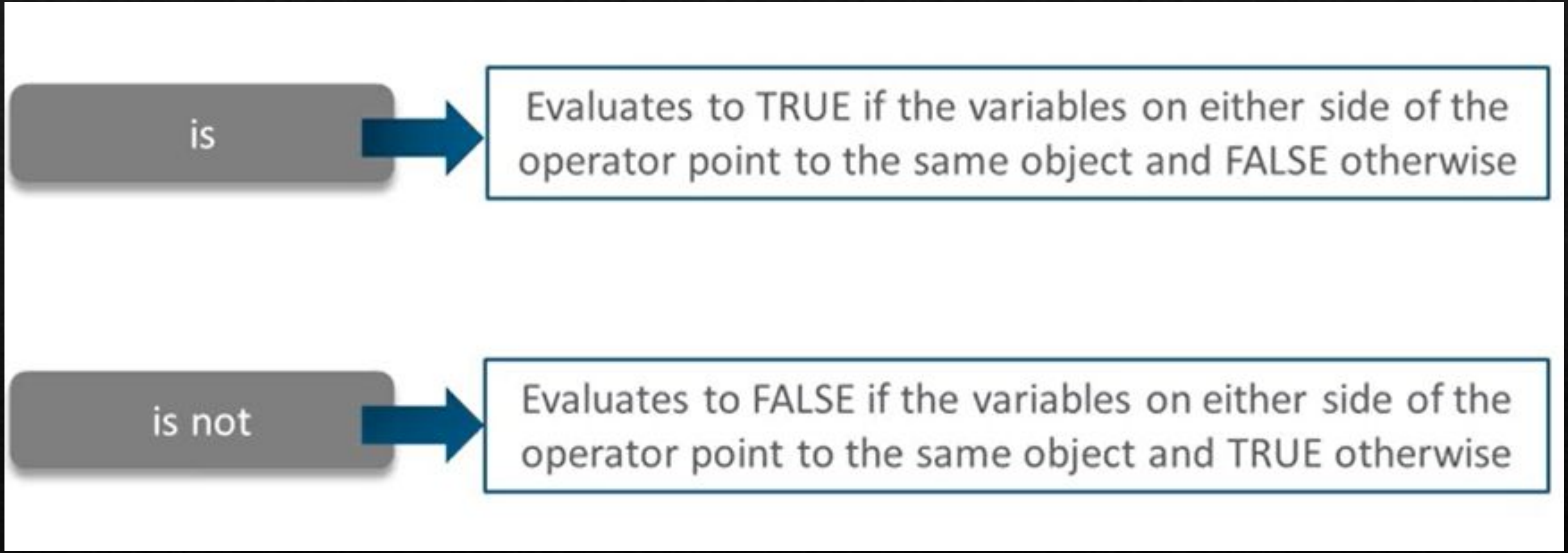


Python Bitwise Operators

Bitwise operators are used to compare (binary) numbers:

Operator	Name	Description
&	AND	Sets each bit to 1 if both bits are 1
|	OR	Sets each bit to 1 if one of two bits is 1
^	XOR	Sets each bit to 1 if only one of two bits is 1
~	NOT	Inverts all the bits
<<	Zero fill left shift	Shift left by pushing zeros in from the right and let the leftmost bits fall off
>>	Signed right shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off

Identity Operator

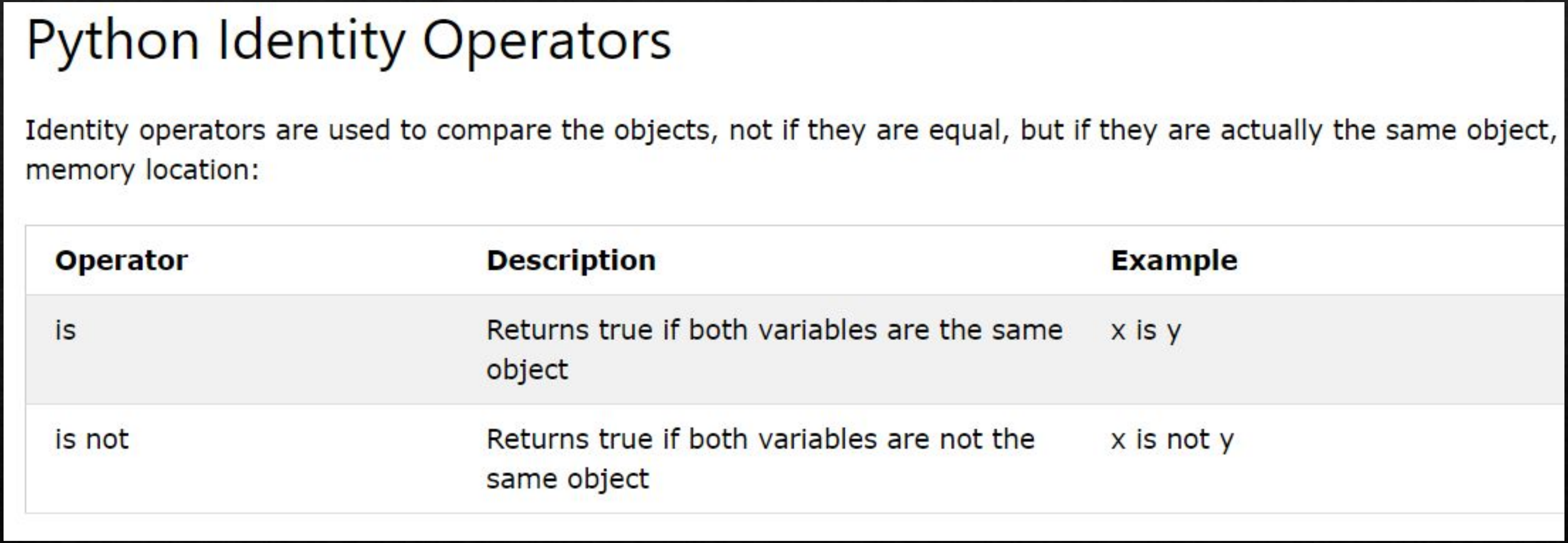


is

Evaluates to TRUE if the variables on either side of the operator point to the same object and FALSE otherwise

is not

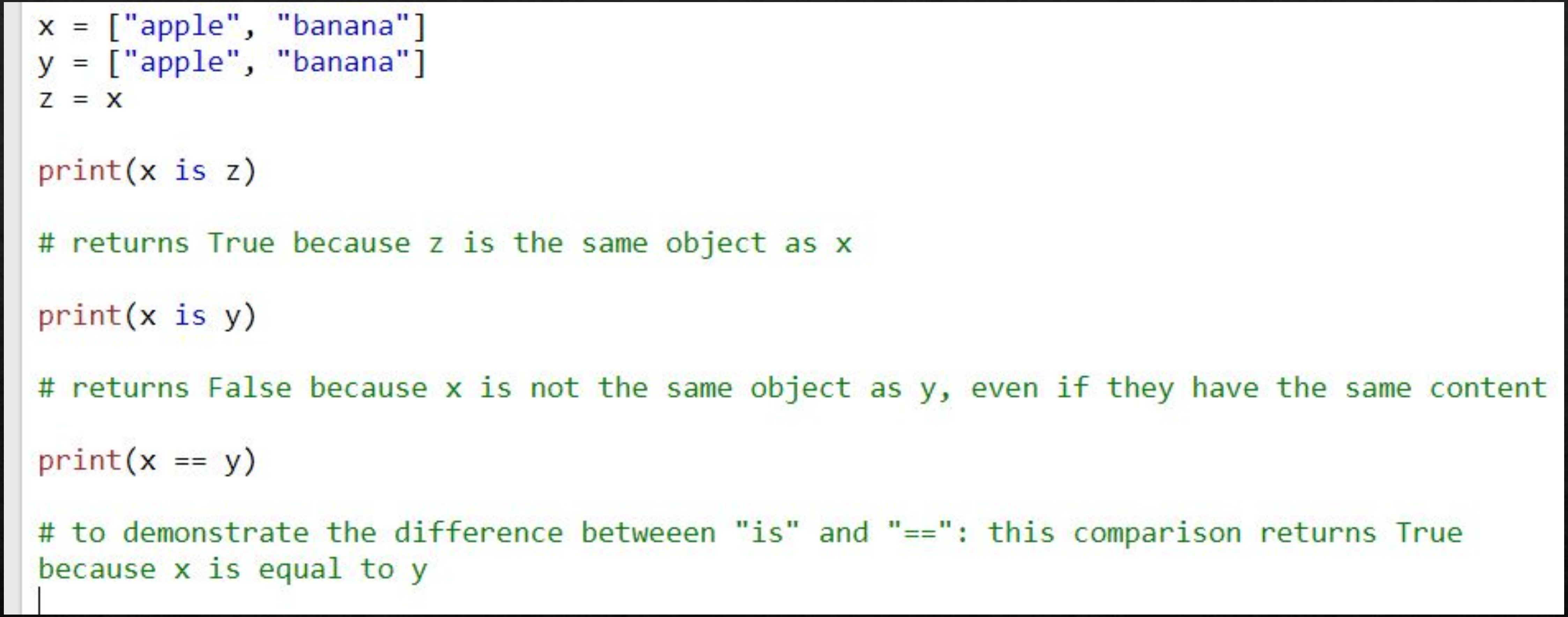
Evaluates to FALSE if the variables on either side of the operator point to the same object and TRUE otherwise



Python Identity Operators

Identity operators are used to compare the objects, not if they are equal, but if they are actually the same object,
memory location:

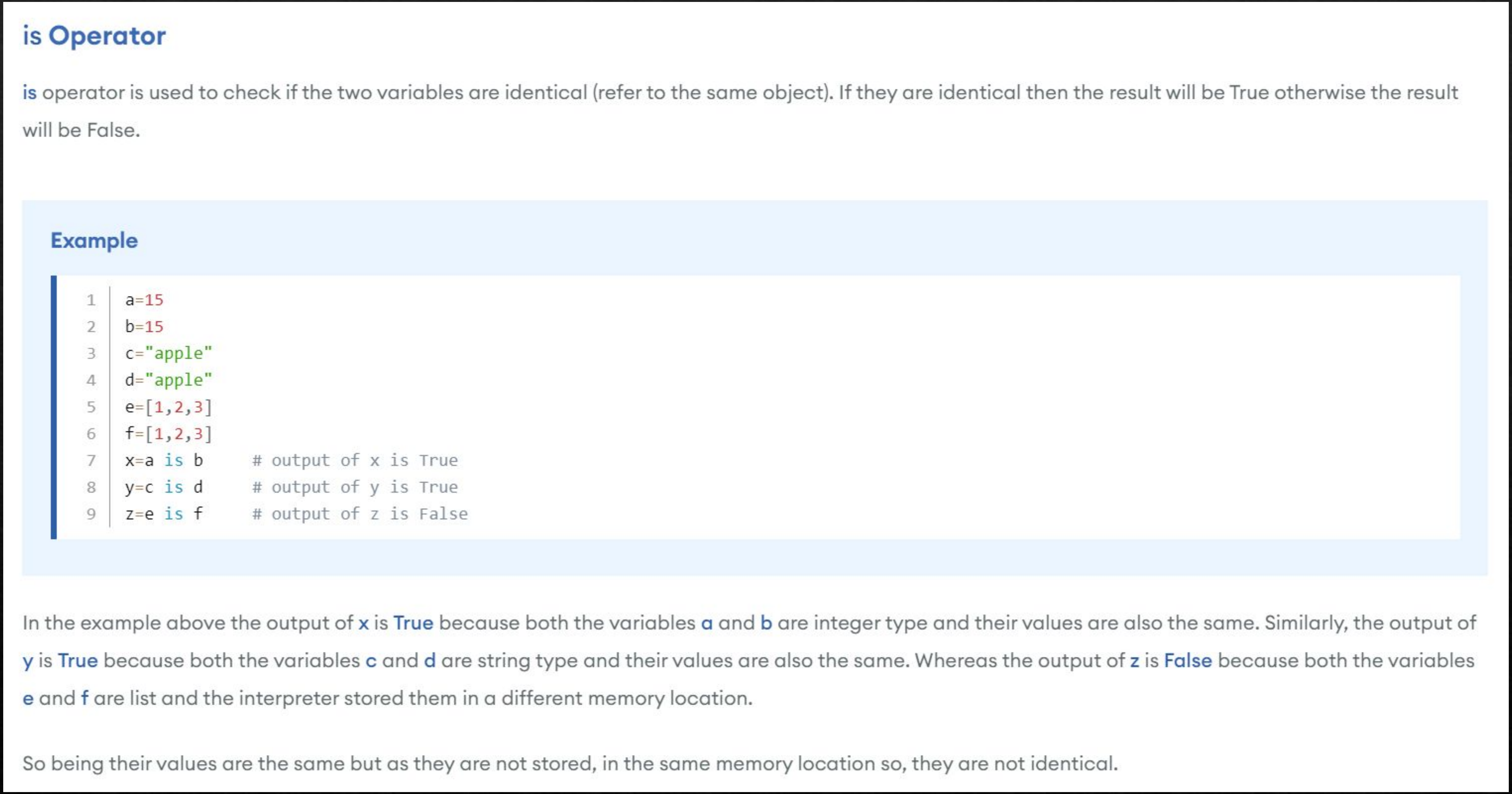
Operator	Description	Example
is	Returns true if both variables are the same object	x is y
is not	Returns true if both variables are not the same object	x is not y



```
x = ["apple", "banana"]
y = ["apple", "banana"]
z = x

print(x is z)

# returns True because z is the same object as x

print(x is y)

# returns False because x is not the same object as y, even if they have the same content

print(x == y)

# to demonstrate the difference betweeen "is" and "==": this comparison returns True
because x is equal to y
```

```
x = [”apple”, ”banana”]
y = [”apple”, ”banana”]
z = x
```

```
print(x is z)
print(x is y)
print(x == y)
```



is Operator

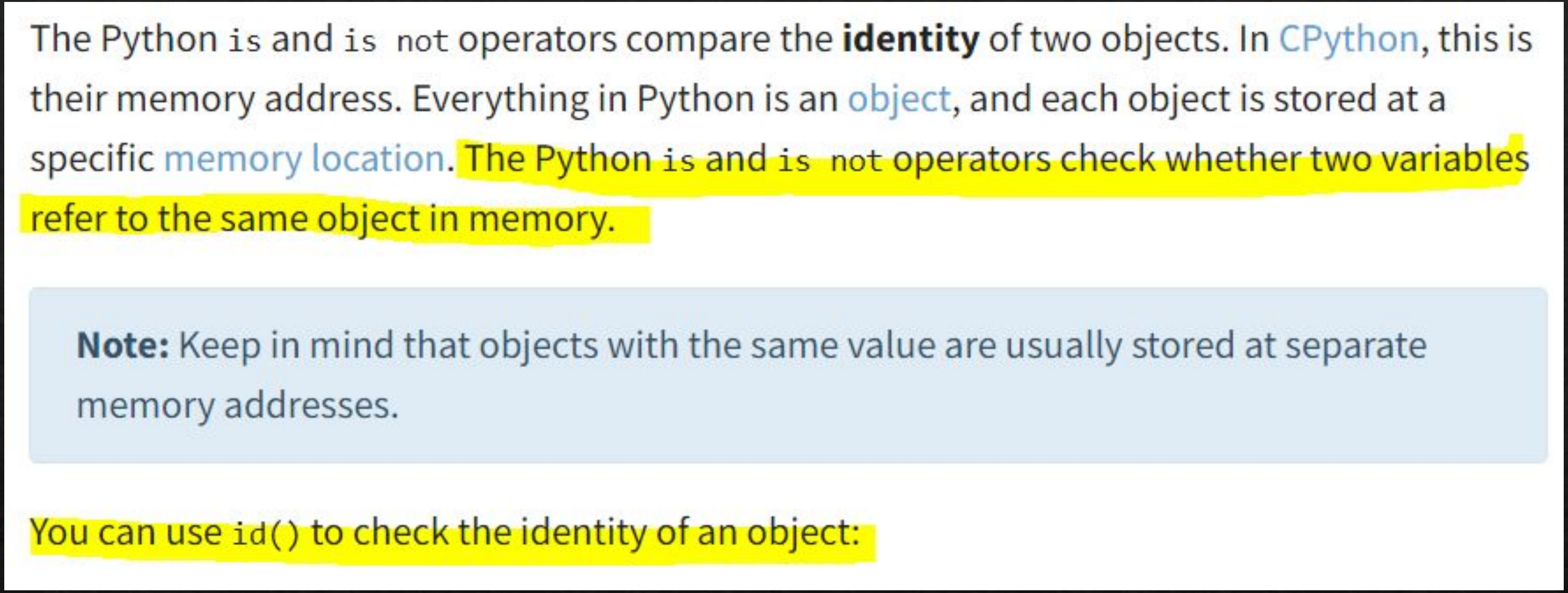
is operator is used to check if the two variables are identical (refer to the same object). If they are identical then the result will be True otherwise the result will be False.

Example

```
a=15
b=15
c="apple"
d="apple"
e=[1,2,3]
f=[1,2,3]
x=a is b     # output of x is True
y=c is d     # output of y is True
z=e is f     # output of z is False
```

In the example above the output of x is True because both the variables a and b are integer type and their values are also the same. Similarly, the output of y is True because both the variables c and d are string type and their values are also the same. Whereas the output of z is False because both the variables e and f are list and the interpreter stored them in a different memory location.

So being their values are the same but as they are not stored, in the same memory location so, they are not identical.

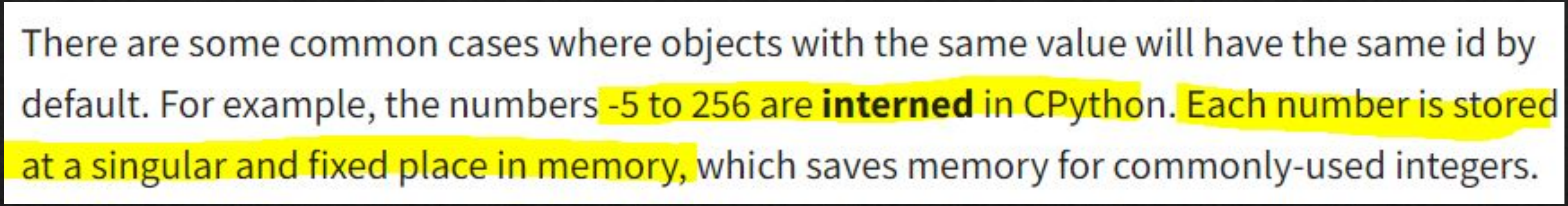

The Python `is` and `is not` operators compare the **identity** of two objects. In CPython, this is their memory address. Everything in Python is an object, and each object is stored at a specific memory location. The Python `is` and `is not` operators check whether two variables refer to the same object in memory.

Note: Keep in mind that objects with the same value are usually stored at separate memory addresses.

You can use `id()` to check the identity of an object:

```
a = 50
b = 50
c= a
print (id(a))
print (id(b))
print (id(c))
print (a is b)
print (c is a)
print (c is b)
```

```
a = 500
b = 500
c= a
print (id(a))
print (id(b))
print (id(c))
print (a is b)
print (c is a)
print (c is b)
```


There are some common cases where objects with the same value will have the same id by default. For example, the numbers -5 to 256 are **interned** in CPython. Each number is stored at a singular and fixed place in memory, which saves memory for commonly-used integers.

```
a = 256
b = 256
print (a is b)
print ( id(a))
print ( id(b))
```

when their values are outside the range of **common integers** (ranging from -5 to 256), they're stored at separate memory addresses.

```
a = 257
b = 257
print (a is b)
print ( id(a))
print ( id(b))
```

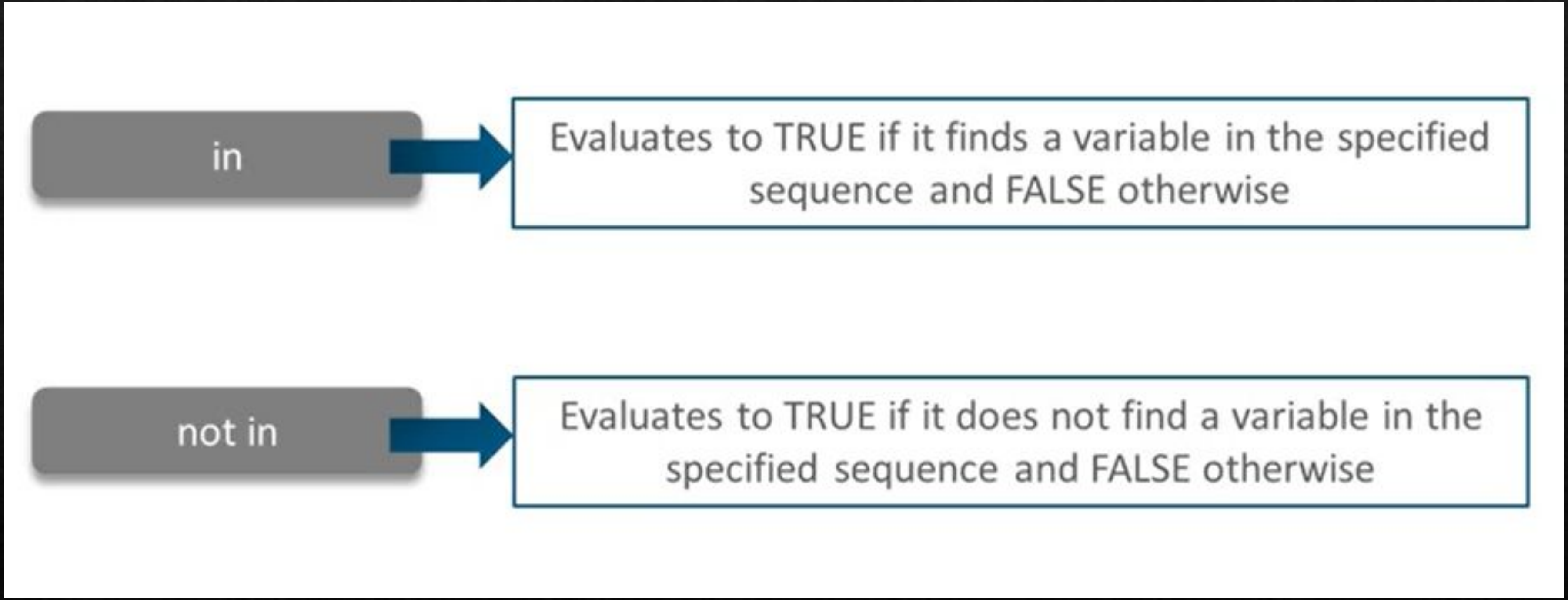
Append

```
a = [1, 2, 3]
b = a
print (a)
print (id(a))
print (b)
print (id(b))
print ()
print (a is b)
print ('*****Append****')
print ()
a.append(4)
print (a)
print (id(a))
print (b)
print (id(b))
print (a is b)
```

Copy- Different Location

```
a = [1, 2, 3]
b = a.copy()
print (a)
print (id(a))
print (b)
print (id(b))
print ()
print (a is b)
print ('*****Append****')
print ()
a.append(4)
print (a)
print (id(a))
print (b)
print (id(b))
print (a is b)
```

Member ship



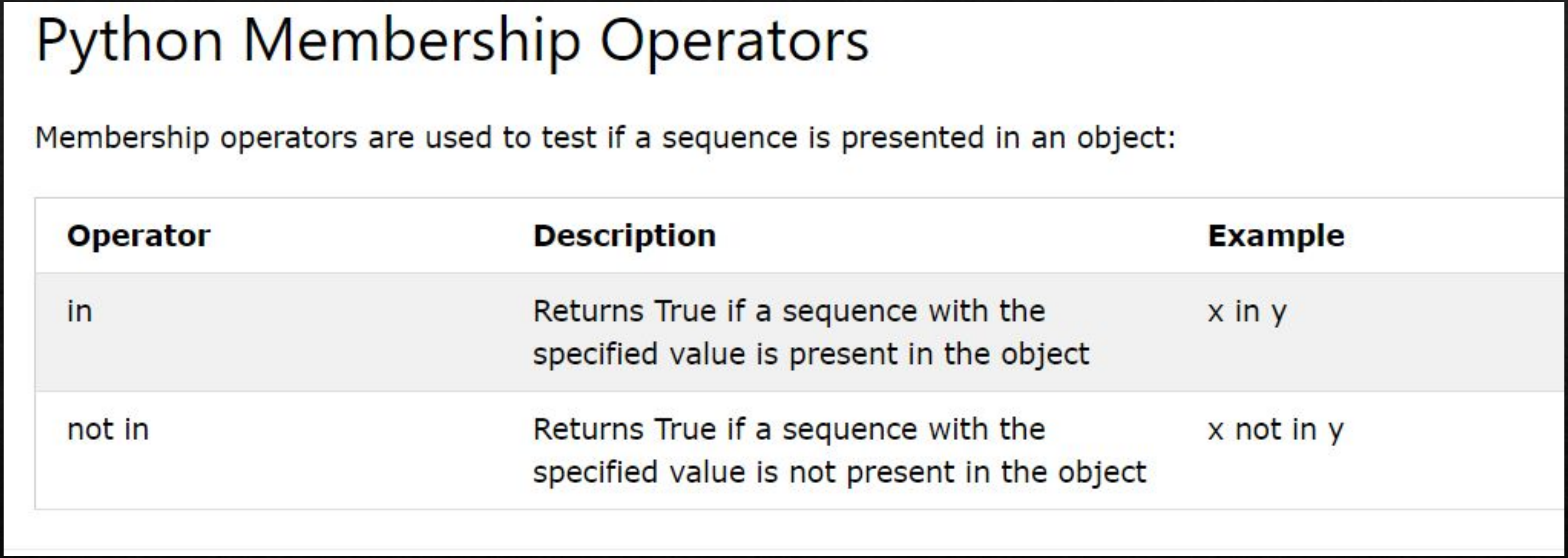
in

Evaluates to TRUE if it finds a variable in the specified sequence and FALSE otherwise

not in

Evaluates to TRUE if it does not find a variable in the specified sequence and FALSE otherwise

```
A = [10,15,20]
print (15 in A)
```



Python Membership Operators

Membership operators are used to test if a sequence is presented in an object:

Operator	Description	Example
in	Returns True if a sequence with the specified value is present in the object	x in y
not in	Returns True if a sequence with the specified value is not present in the object	x not in y