Python Operators

Operators are used to perform operations on variables and values.

In the example below, we use the + operator to add together two values:

```
In [1]: print(10 + 5)
```

```
15
```

Python divides the operators in the following groups:

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators
- Identity operators
- Membership operators
- Bitwise operators

Python Arithmetic Operators

Arithmetic operators are used with numeric values to perform common mathematical operations:

`+`	Addition	`x + y`
`-`	Subtraction	`x - y`
`*`	Multiplication	`x * y`
`/`	Division	`x / y`
`%`	Modulus	`x % y`
`**`	Exponentiation	`x ** y`
`//`	Floor division	`x // y`

Python Assignment Operators

Assignment operators are used to assign values to variables:

`=`	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

Python Comparison Operators

Comparison operators are used to compare two values:

Operator	Name	Example	Try it
==	Equal	x == y	Try it »
!=	Not equal	x != y	Try it »
>	Greater than	x > y	Try it »
<	Less than	x < y	Try it »
>=	Greater than or equal to	x >= y	Try it »
<=	Less than or equal to	x <= y	Try it »

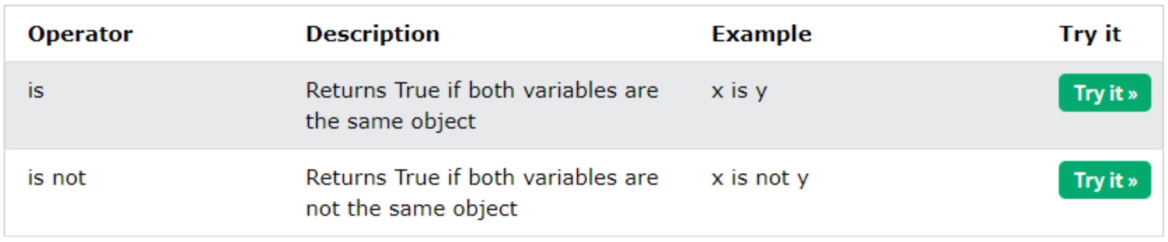
Python Logical Operators

Logical operators are used to combine conditional statements:

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

Python Identity Operators

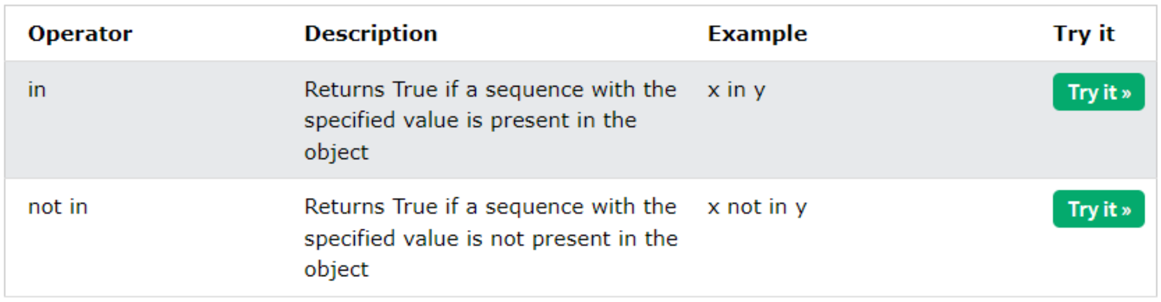
Identity operators are used to compare the objects, not if they are equal, but if they are actually the same object, with the same memory location:



Operator	Description	Example	Try it
is	Returns True if both variables are the same object	x is y	Try it »
is not	Returns True if both variables are not the same object	x is not y	Try it »

Python Membership Operators

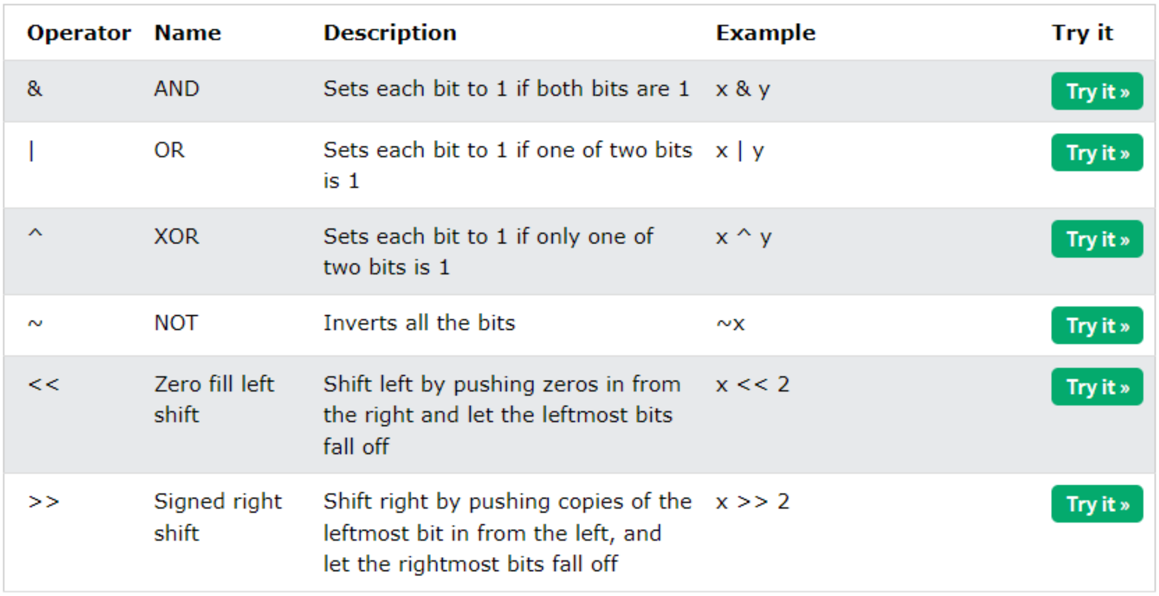
Membership operators are used to test if a sequence is presented in an object:



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

Python Bitwise Operators

Bitwise operators are used to compare (binary) numbers:



Operator	Name	Description	Example	Try it
&	AND	Sets each bit to 1 if both bits are 1	x & y	Try it »
|	OR	Sets each bit to 1 if one of two bits is 1	x | y	Try it »
^	XOR	Sets each bit to 1 if only one of two bits is 1	x ^ y	Try it »
~	NOT	Inverts all the bits	~x	Try it »
<<	Zero fill left shift	Shift left by pushing zeros in from the right and let the leftmost bits fall off	x << 2	Try it »
>>	Signed right shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off	x >> 2	Try it »

In [8]:
```python
print(6 & 3)



# """
# The & operator compares each bit and set it to 1 if both are 1,
# otherwise it is set to 0:

# 6 = 0000000000000110
# 3 = 0000000000000011
# --------------------
# 2 = 0000000000000010
# ====================

# Decimal numbers and their binary values:
# 0 = 0000000000000000
# 1 = 0000000000000001
# 2 = 0000000000000010
# 3 = 0000000000000011
# 4 = 0000000000000100
# 5 = 0000000000000101
# 6 = 0000000000000110
# 7 = 0000000000000111
# """
```

```
2
```

In [9]:
```
print(6 | 3)



# """
# The | operator compares each bit and set it to 1 if one or both is 1,
# otherwise it is set to 0:

# 6 = 0000000000000110
# 3 = 0000000000000011
# --------------------
# 7 = 0000000000000111
# ====================

# Decimal numbers and their binary values:
# 0 = 0000000000000000
# 1 = 0000000000000001
# 2 = 0000000000000010
# 3 = 0000000000000011
# 4 = 0000000000000100
# 5 = 0000000000000101
# 6 = 0000000000000110
# 7 = 0000000000000111
# """
```

```
7
```

In [11]:
```
print(6 ^ 3)



# """
# The ^ operator compares each bit and set it to 1 if only one is 1,
# otherwise (if both are 1 or both are 0) it is set to 0:

# 6 = 0000000000000110
# 3 = 0000000000000011
# --------------------
# 5 = 0000000000000101
# ====================

# Decimal numbers and their binary values:
# 0 = 0000000000000000
# 1 = 0000000000000001
# 2 = 0000000000000010
# 3 = 0000000000000011
# 4 = 0000000000000100
# 5 = 0000000000000101
# 6 = 0000000000000110
# 7 = 0000000000000111
# """
```

```
5
```

In [13]:
```python
print(~3)



# """
# The ~ operator inverts each bit (0 becomes 1 and 1 becomes 0).

# Inverted 3 becomes -4:
#  3 = 0000 0000 0000 0011
# -4 = 1111 1111 1111 1100

# Decimal numbers and their binary values:
#  4 = 0000000000000100
#  3 = 0000000000000011
#  2 = 0000000000000010
#  1 = 0000000000000001
#  0 = 0000000000000000
# -1 = 1111111111111111
# -2 = 1111111111111110
# -3 = 1111111111111101
# -4 = 1111111111111100
# """
```

```
-4
```

In [6]:
```
print(3 << 2)



# """
# The << operator inserts the specified number of 0's (in this case 2)
# from the right and let the same amount of leftmost bits fall off:

# If you push 00 in from the left:
#  3 = 0000000000000011
# becomes
# 12 = 0000000000001100

# Decimal numbers and their binary values:
#  0 = 0000000000000000
#  1 = 0000000000000001
#  2 = 0000000000000010
#  3 = 0000000000000011
#  4 = 0000000000000100
#  5 = 0000000000000101
#  6 = 0000000000000110
#  7 = 0000000000000111
#  8 = 0000000000001000
#  9 = 0000000000001001
# 10 = 0000000000001010
# 11 = 0000000000001011
# 12 = 0000000000001100
# """
```

```
12
```

In [16]:
```
print(8 >> 2)  #0010


# """
# The >> operator moves each bit the specified number of times to the right. Er

# If you move each bit 2 times to the right, 8 becomes 2:
#  8 = 0000000000001000
# becomes
#  2 = 0000000000000010

# Decimal numbers and their binary values:
#  0 = 0000000000000000
#  1 = 0000000000000001
#  2 = 0000000000000010
#  3 = 0000000000000011
#  4 = 0000000000000100
#  5 = 0000000000000101
#  6 = 0000000000000110
#  7 = 0000000000000111
#  8 = 0000000000001000
#  9 = 0000000000001001
# 10 = 0000000000001010
# 11 = 0000000000001011
# 12 = 0000000000001100
# """
```

```
2
```

Operator Precedence

Operator precedence describes the order in which operations are performed.

In [17]:
```
# Parentheses has the highest precedence,
# meaning that expressions inside parentheses must be evaluated first:

print((6 + 3) - (6 + 3))
```

```
0
```

In [18]:
```
# Multiplication * has higher precedence than addition +,
# and therefor multiplications are evaluated before additions:

print(100 + 5 * 3)
```

```
115
```

The precedence order is described in the table below, starting with the highest precedence at the top:

Operator	Description	Try it
`()`	Parentheses	Try it »
`**`	Exponentiation	Try it »
`+x` `-x` `~x`	Unary plus, unary minus, and bitwise NOT	Try it »
`*` `/` `//` `%`	Multiplication, division, floor division, and modulus	Try it »
`+` `-`	Addition and subtraction	Try it »
`<<` `>>`	Bitwise left and right shifts	Try it »
`&`	Bitwise AND	Try it »
`^`	Bitwise XOR	Try it »
`|`	Bitwise OR	Try it »
`==` `!=` `>` `>=` `<` `<=` `is` `is not` `in` `not in`	Comparisons, identity, and membership operators	Try it »
`not`	Logical NOT	Try it »
`and`	AND	Try it »
`or`	OR	Try it »

In [ ]:

In [ ]: