Python programming for beginners

Stefan Zhauryd
Instructor

Module 3

Boolean Values, Conditional Execution, Loops, Lists and List Processing, Logical and Bitwise Operations

In this module, you will learn about:

- **the Boolean data type;**
- **relational operators;**
- **making decisions in Python (if, if-else, if-elif, else)**
- **how to repeat code execution using loops (while, for)**
- **how to perform logic and bitwise operations in Python;**
- lists in Python (constructing, indexing, and slicing; content manipulation)
- how to sort a list using bubble-sort algorithms;
- multidimensional lists and their applications.

Python supports the usual logical conditions from mathematics:

```
b = 0
a = 2
b += 1
```

- Equals: a **==** b
- Not Equals: a **!=** b
- Less than: a **<** b
- Less than or equal to: a **<=** b
- Greater than: a **>** b
- Greater than or equal to: a **>=** b

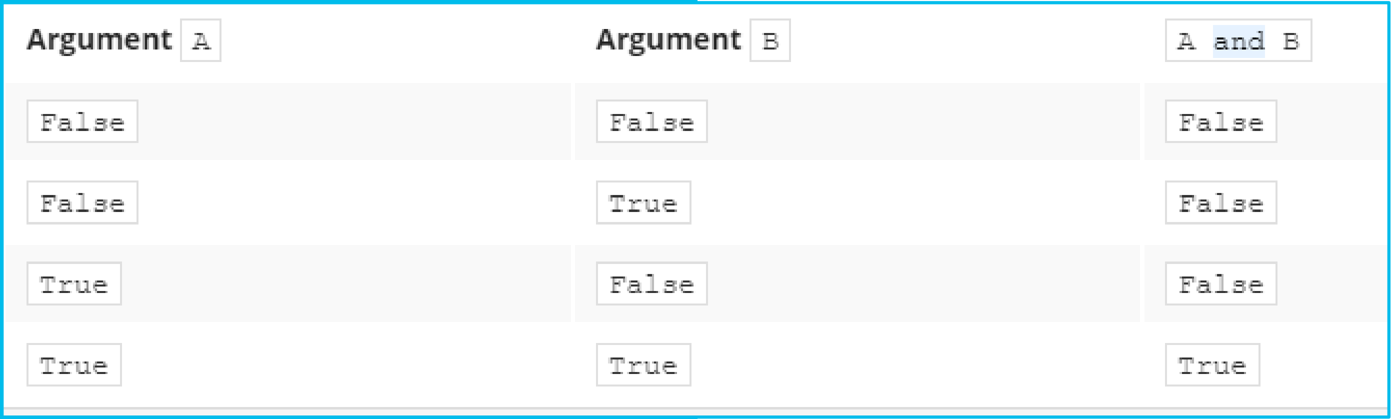
Computer logic
and or not

Try to use:

```
A = True
B = False
print(A and B, type(A))
```

One logical **conjunction operator** in Python is the word and. It's a binary operator with a priority that is lower than the one expressed by the comparison operators.

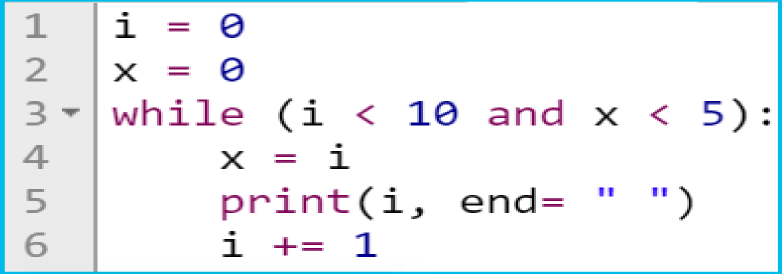
Computer logic **and**



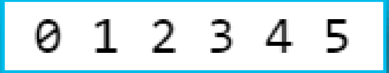
Argument `A`	Argument `B`	`A and B`
`False`	`False`	`False`
`False`	`True`	`False`
`True`	`False`	`False`
`True`	`True`	`True`

```
a = True
print(a, type(a)) #<class 'bool'>
print(not a) #False
```

Computer logic **and**

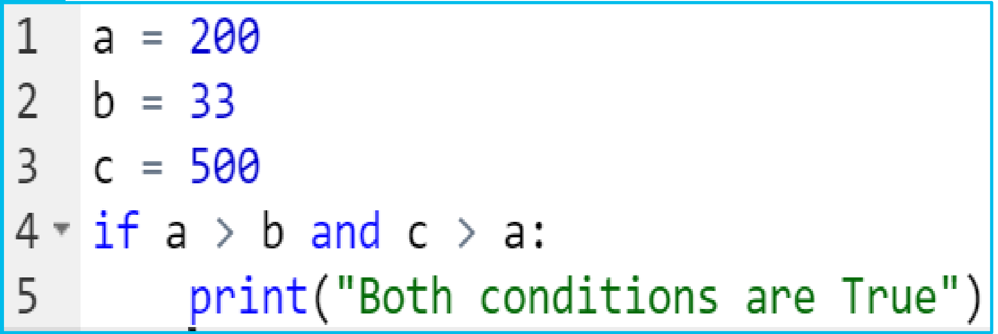


```
i = 0
x = 0
while (i < 10 and x < 5):
    x = i
    print(i, end= " ")
    i += 1
```

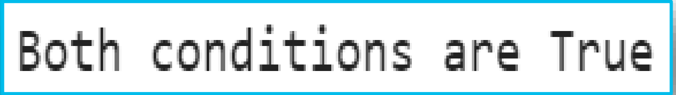


```
0 1 2 3 4 5
```

One logical conjunction operator in Python is the word and. It's a binary operator with a priority that is lower than the one expressed by the comparison operators.



```
a = 200
b = 33
c = 500
if a > b and c > a:
    print("Both conditions are True")
```



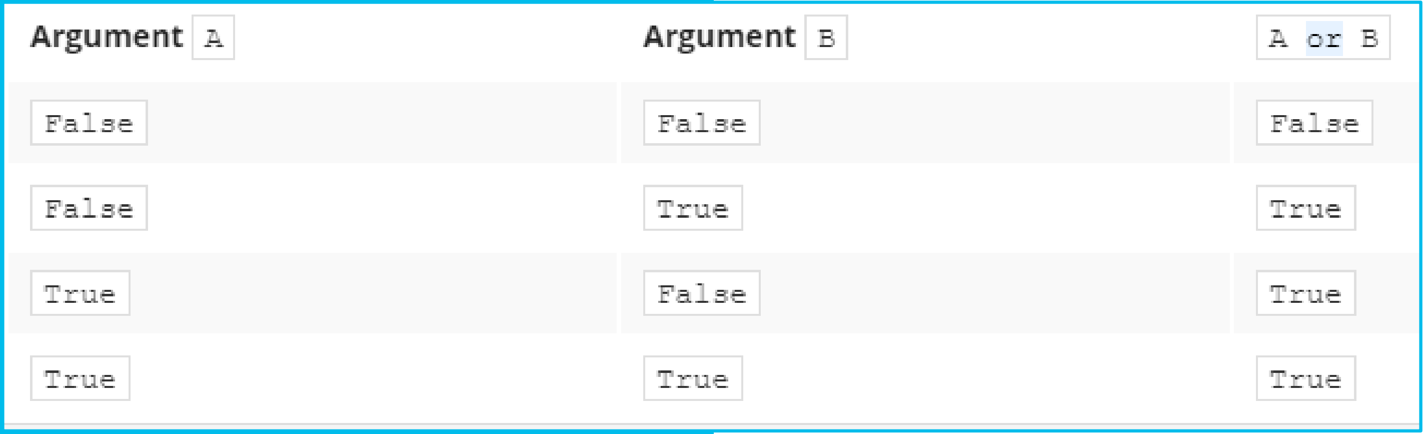
```
Both conditions are True
```

Try to use:

```
A = True

B = False

print(A or B, type(A))
```

Computer logic
or

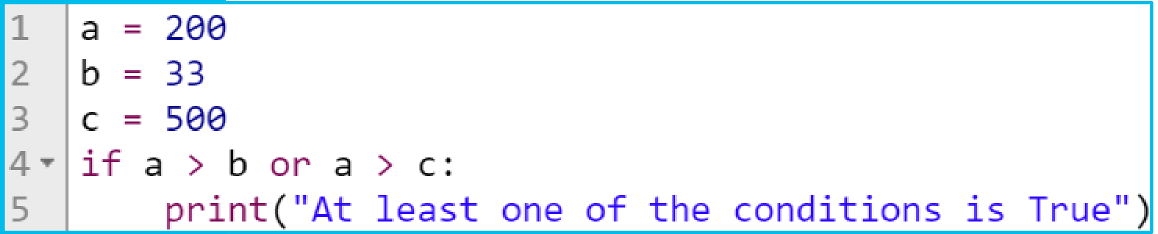
A **disjunction operator** is the word or. It's a binary operator with a lower priority than and (just like + compared to *). Its truth table is as follows:



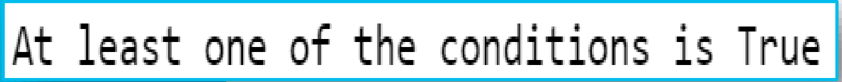
Argument `A`	Argument `B`	`A or B`
`False`	`False`	`False`
`False`	`True`	`True`
`True`	`False`	`True`
`True`	`True`	`True`

Computer logic
or

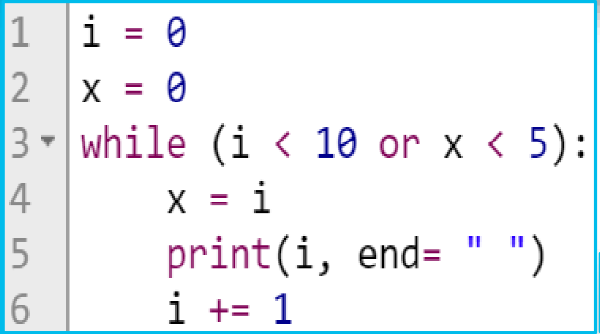
A disjunction operator is the word or. It's a binary operator with a lower priority than and (just like + compared to *). Its truth table is as follows:



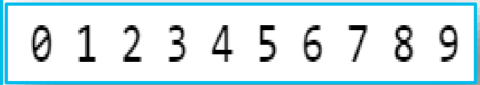
```python
a = 200
b = 33
c = 500
if a > b or a > c:
    print("At least one of the conditions is True")
```



```
At least one of the conditions is True
```



```python
i = 0
x = 0
while (i < 10 or x < 5):
    x = i
    print(i, end= " ")
    i += 1
```



```
0 1 2 3 4 5 6 7 8 9
```

Try to use:

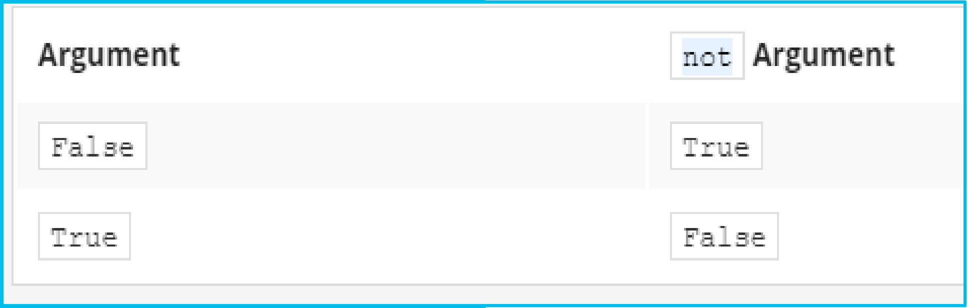
```
B = False

print(type(B), not B)
```

Computer logic
not

```
 a = True
print(a, type(a), not a)
#True <class 'bool'> False
```

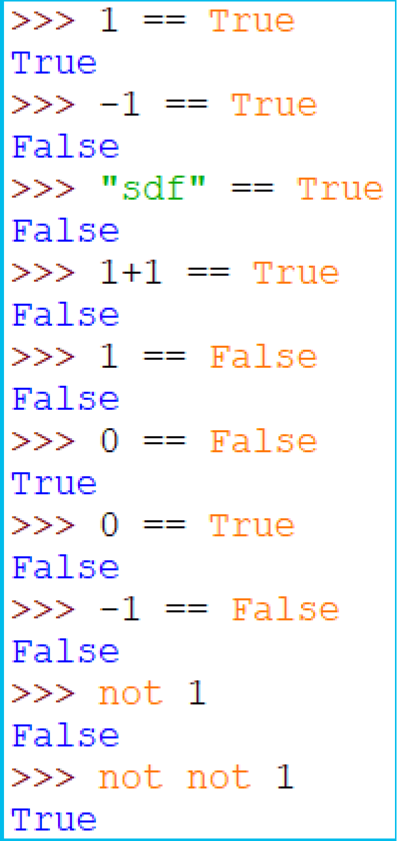
In addition, there's another operator that can be applied for constructing conditions. It's a unary operator performing a logical negation. Its operation is simple: it turns truth into falsehood and falsehood into truth.

This operator is written as the word not, and its priority is very high: the same as the unary + and -. Its truth table is simple:



Argument	`not` Argument
`False`	`True`
`True`	`False`

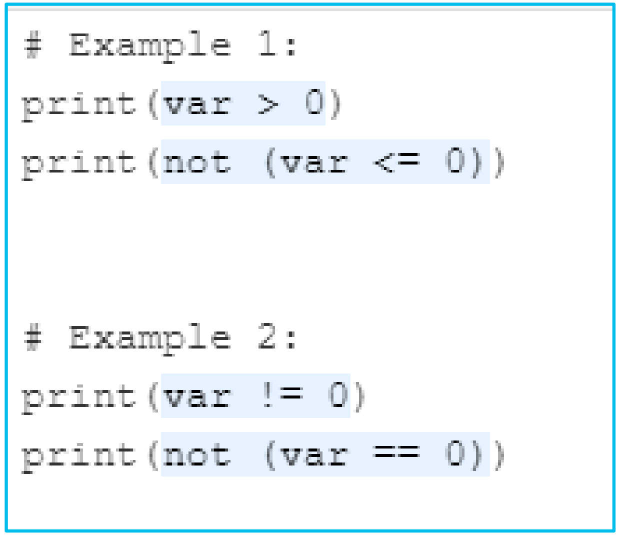
Logical expressions



```
>>> 1 == True
True
>>> -1 == True
False
>>> "sdf" == True
False
>>> 1+1 == True
False
>>> 1 == False
False
>>> 0 == False
True
>>> 0 == True
False
>>> -1 == False
False
>>> not 1
False
>>> not not 1
True
```

You may be familiar with De Morgan's laws. They say that:

- The negation of a conjunction is the disjunction of the negations.
- The negation of a disjunction is the conjunction of the negations.

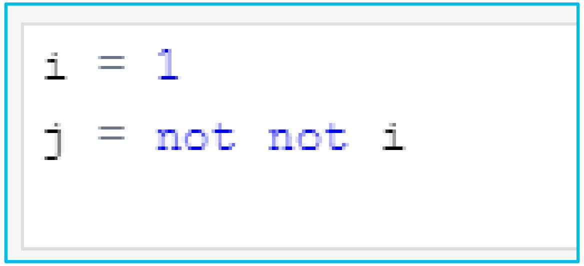


```
# Example 1:
print(var > 0)
print(not (var <= 0))


# Example 2:
print(var != 0)
print(not (var == 0))
```

Logical values vs. single bits

Logical operators take their arguments as a whole regardless of how many bits they contain. The operators are aware only of the value: zero (when all the bits are reset) means False; not zero (when at least one bit is set) means True.

The result of their operations is one of these values: False or True. This means that this snippet will assign the value True to the j variable if i is not zero; otherwise, it will be False.



```
i = 1
j = not not i
```

Home work 2.7

*exit by 'exit' word
choose +-/ and so on
•Comments ###

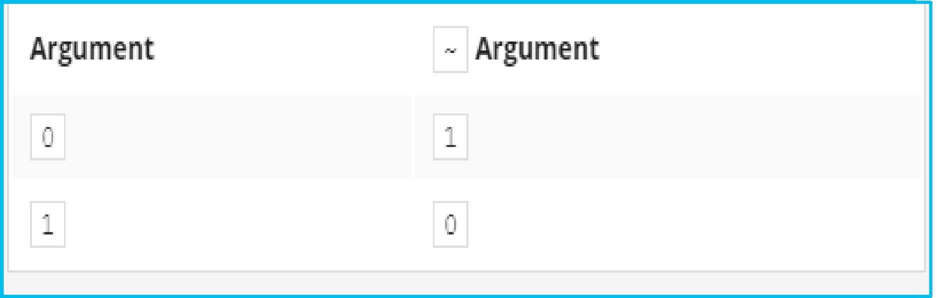
Write the **calculator**.

Operations: **+ - / * ** // %**

- Use the input() to get type of operation and data from user.
- input() int() float() type() and so on. use if/elif/else, for/while

Just investigate, don't afraid of mistakes https://ru.wikipedia.org/wiki/Бит

https://ru.wikipedia.org/wiki/Битовая_операция



Argument	~ Argument
0	1
1	0

Bitwise operators

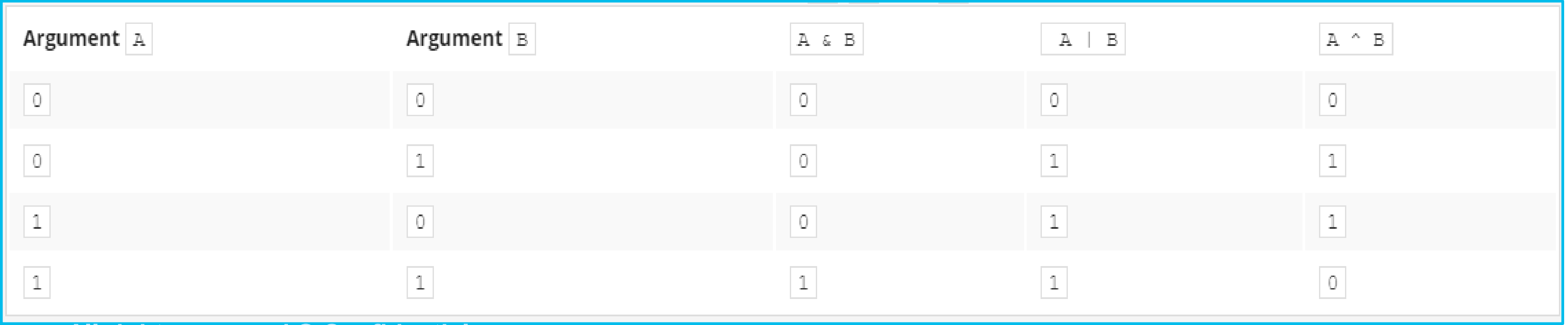
Here are all of them:

& (ampersand) - bitwise conjunction;

| (bar) - bitwise disjunction;

~ (tilde) - bitwise negation;

^ (caret) - bitwise exclusive or (xor).



Argument A	Argument B	A & B	A | B	A ^ B
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

https://www.bestprog.net/ru/2019/10/21/python-bitwise-operators-ru/

http://mf.grsu.by/UchProc/livak/b_org/oal_5.htm

Bitwise operators

Let's make it easier:

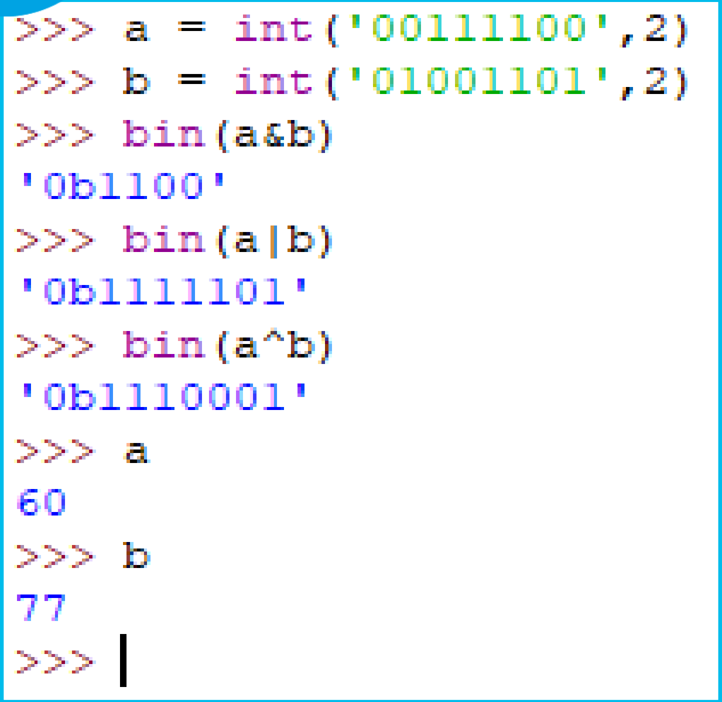
- & requires **exactly two 1s** to provide 1 as the result;
- | requires **at least one 1** to provide 1 as the result;
- ^ requires **exactly one 1** to provide 1 as the result.
- ~1 = 0 ~0 = 1

Bitwise operators

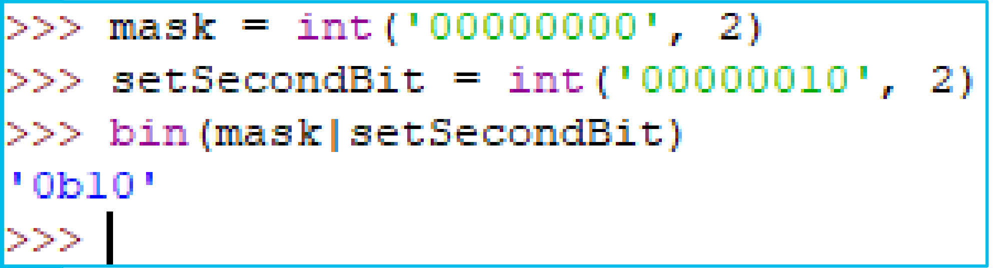
Let us add an important remark: the arguments of these operators must be integers; we must not use floats here.

The difference in the operation of the logical and bit operators is important: the logical operators do not penetrate into the bit level of its argument. They're only interested in the final integer value.

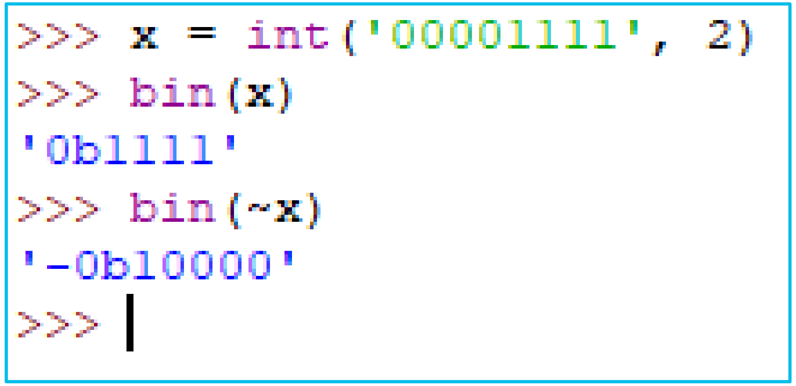
Bitwise operators are stricter: they deal with every bit separately. If we assume that the integer variable occupies 64 bits (which is common in modern computer systems), you can imagine the bitwise operation as a 64-fold evaluation of the logical operator for each pair of bits of the arguments. This analogy is obviously imperfect, as in the real world all these 64 operations are performed at the same time (simultaneously).



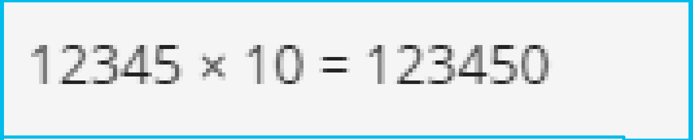
```
>>> a = int('00111100',2)
>>> b = int('01001101',2)
>>> bin(a&b)
'0b1100'
>>> bin(a|b)
'0b1111101'
>>> bin(a^b)
'0b1110001'
>>> a
60
>>> b
77
>>>
```

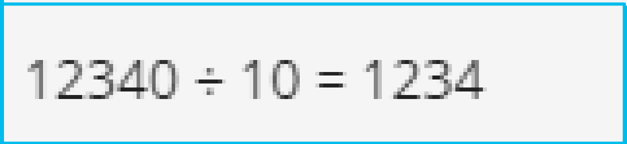


```
>>> mask = int('00000000', 2)
>>> setSecondBit = int('00000010', 2)
>>> bin(mask|setSecondBit)
'0b10'
>>>
```

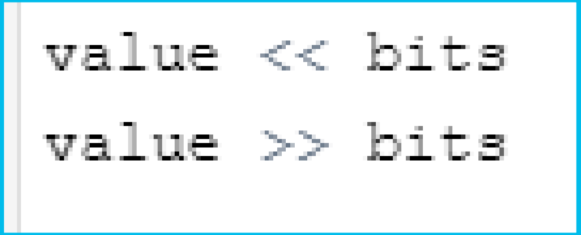


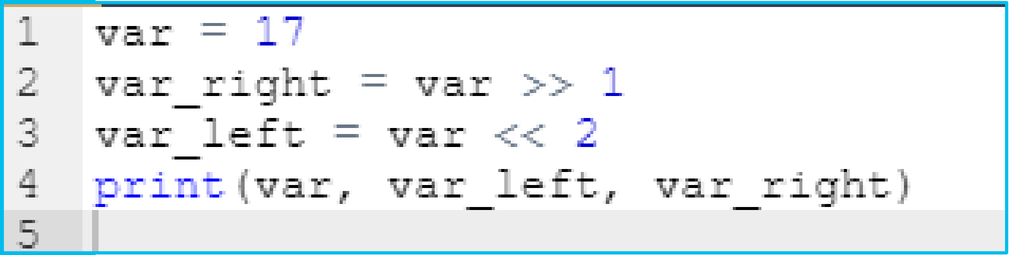
```
>>> x = int('00001111', 2)
>>> bin(x)
'0b1111'
>>> bin(~x)
'-0b10000'
>>>
```

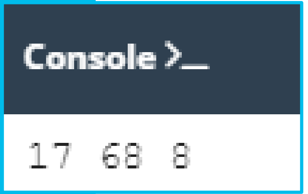

12345 × 10 = 123450

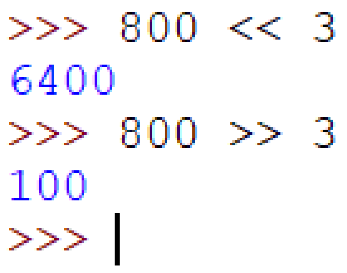

12340 ÷ 10 = 1234

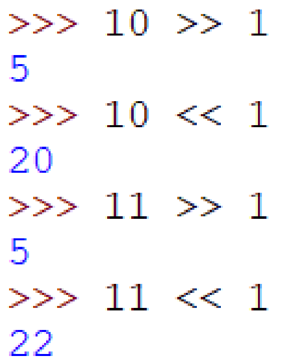
Binary left shift and binary right shift


```
value << bits
value >> bits
```


```
var = 17
var_right = var >> 1
var_left = var << 2
print(var, var_left, var_right)
```


```
Console >_
17 68 8
```


```
>>> 800 << 3
6400
>>> 800 >> 3
100
>>>
```


```
>>> 10 >> 1
5
>>> 10 << 1
20
>>> 11 >> 1
5
>>> 11 << 1
22
```

Note:

17 >> 1 → 17 // 2 (17 floor-divided by 2 to the power of 1) → 8 (shifting to the right by one bit is the same as integer division by two)

17 << 2 → 17 * 4 (17 multiplied by 2 to the power of 2) → 68 (shifting to the left by two bits is the same as integer multiplication by four)

Key takeaways

1. Python supports the following **logical operators:**

- and → if both operands are true, the condition is true, e.g., (True and True) is True,
- or → if any of the operands are true, the condition is true, e.g., (True or False) is True,
- not → returns false if the result is true, and returns true if the result is false, e.g., not True is False.

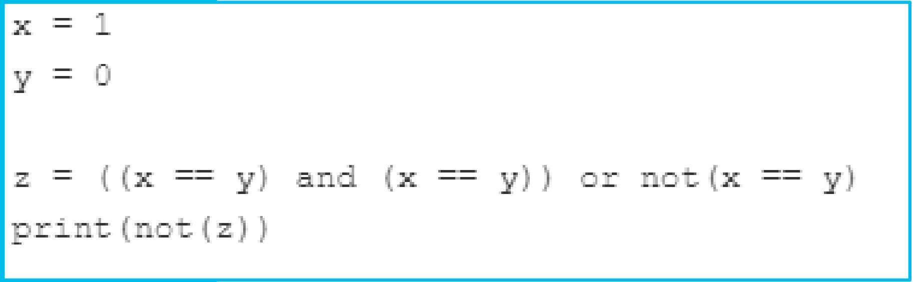
2. You can use **bitwise operators** to manipulate single bits of data. The following sample data:

- x = 15, which is 0000 1111 in binary,
- y = 16, which is 0001 0000 in binary.
- will be used to illustrate the meaning of bitwise operators in Python.

Analyze the examples below:

- & does a bitwise and, e.g., x & y = 0, which is 0000 0000 in binary,
- | does a bitwise or, e.g., x | y = 31, which is 0001 1111 in binary,
- ˜ does a bitwise not, e.g., ˜ x = 240*, which is 1111 0000 in binary,
- ^ does a bitwise xor, e.g., x ^ y = 31, which is 0001 1111 in binary,
- >> does a bitwise right shift, e.g., y >> 1 = 8, which is 0000 1000 in binary,
- << does a bitwise left shift, e.g., y << 3 = , which is 1000 0000 in binary,

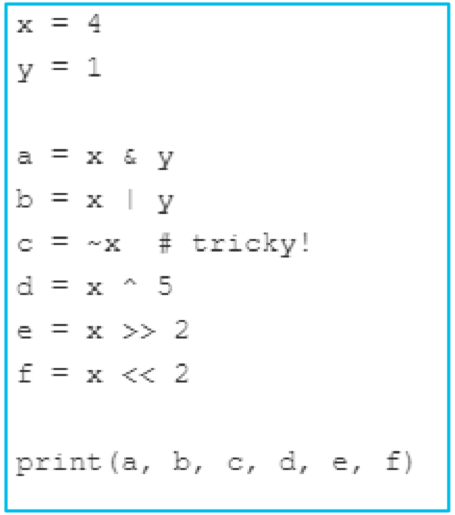
Examples



```
x = 1
y = 0

z = ((x == y) and (x == y)) or not(x == y)
print(not(z))
```

output:

```
False
```



```
x = 4
y = 1

a = x & y
b = x | y
c = ~x  # tricky!
d = x ^ 5
e = x >> 2
f = x << 2

print(a, b, c, d, e, f)
```

output:

```
0 5 -5 1 1 16
```

ЗАДАНИЯ

1) Прорешать всю классную работу
2) Выполнить все домашние задания

Почитать:

1) Byte of Python

Прочитать страницы -

стр. 47-54

стр. 55-63

Крайний срок сдачи 30/09 в 21:00 (можно раньше, но не позже)

ЗАДАНИЯ

Название файлов, которые вы отправляете мне в telegram:

Vasia_Pupkin_class_work_L3_0.py

+все задания ОДНИМ ФАЙЛОМ - Vasia_Pupkin_L3_0.py

Формат сообщения которое вы присылаете мне

(после полного выполнения домашнего задания, только один раз) в Telegram:

Добрый день/вечер.

Я Вася Пупкин, и это мои домашние задания к лекции 3 часть 0.

И отправляете файлы

Крайний срок сдачи 30/09 в 21:00 (можно раньше, но не позже)

https://docs.github.com/articles/using-pull-requests

Q&A

Create your
possibilities.
Bye bye.