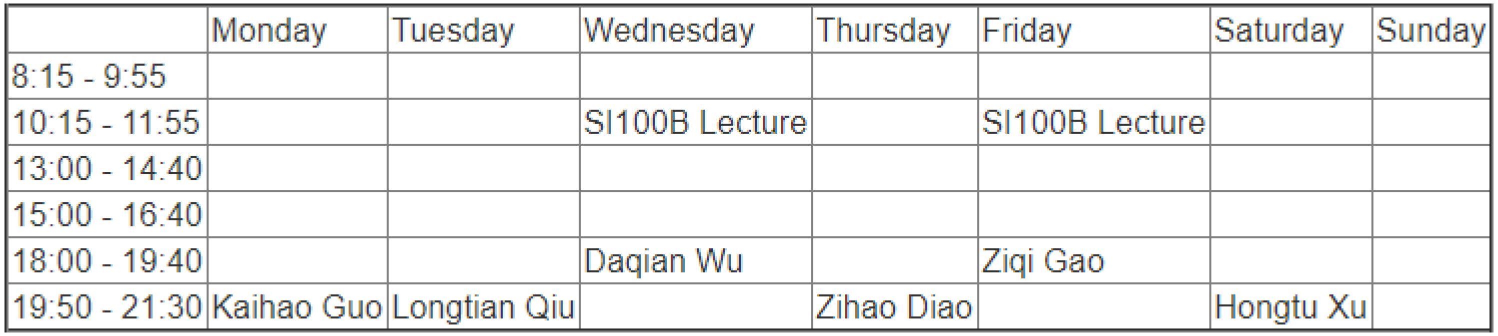
SI100B Introduction to Information Science and Technology
Python Programming

张海鹏 Haipeng Zhang
School of Information Science and Technology
ShanghaiTech University

TA office hours

- **Start today, first 4 weeks**
- **Location: SIST 1B-101**



	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday	Sunday
8:15 - 9:55							
10:15 - 11:55			SI100B Lecture		SI100B Lecture		
13:00 - 14:40							
15:00 - 16:40							
18:00 - 19:40			Daqian Wu		Ziqi Gao		
19:50 - 21:30	Kaihao Guo	Longtian Qiu		Zihao Diao		Hongtu Xu	

Tutorial

- Tutorial 1 video recording has been uploaded, check Piazza
- Tutorial time will be decided soon, according to the pool results.

Learning Objectives

- Variables
- Expressions
- Statements

Constants

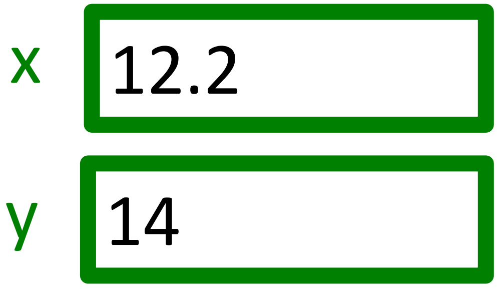
- Fixed values such as numbers, letters, and strings are called “constants” because their value does not change.
- Numeric constants: e.g. 100 280 8.8
- String constants use single quotes (‘) or double quotes (“)

```
>>> print(123)
123
>>> print(98.6)
98.6
>>> print('Hello world’)
Hello world
```

Variables

- A variable is a named place in the memory where a programmer can store data and later retrieve the data using the variable “name”.
- Programmers need to choose the names of the variables.
- You can change the contents of a variable in a later statement.

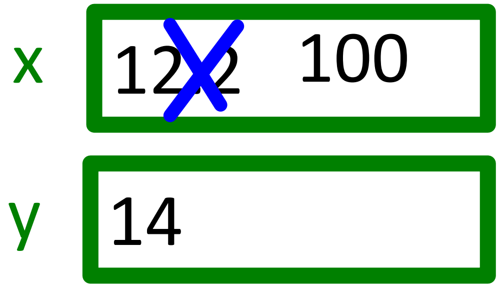
```
x = 12.2
y = 14
```



Variables

- A variable is a named place in the memory where a programmer can store data and later retrieve the data using the variable “name”.
- Programmers need to choose the names of the variables.
- You can change the contents of a variable in a later statement.

```
x = 12.2
y = 14
x = 100
```



Variable name rules

1. Must start with a letter or underscore _
2. Only consist of letters, numbers, and underscores
3. Case sensitive
   - ❑ Different: smith Smith SMITH SmiTH
4. You can not use reserved words as variable names

   smith $smiths smith23 _smith _23_

   23smith smith.23 a+b smiTH -smith

Variable name rules

1. Must start with a letter or underscore _
2. Only consist of letters, numbers, and underscores
3. Case sensitive
   - ❑ Different: smith Smith SMITH SmiTH
4. You can not use reserved words as variable names

   smith $smiths smith23 _smith _23_

   23smith smith.23 a+b smiTH -smith

Green: Good var names **Red: Bad var names**

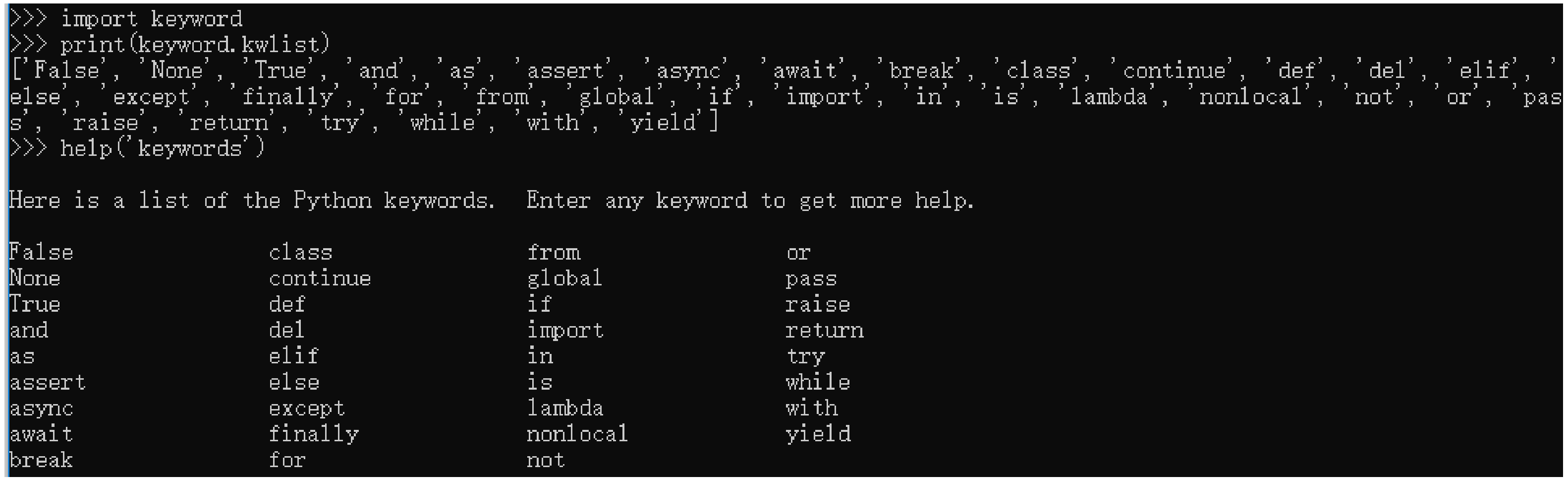
Reserved words

and del for is raise assert elif from lambda return break else global not try class except if or while continue exec import pass yield def finally in print as with

```
>>> 32smith = ‘hello world’
SyntaxError: invalid syntax

>>> class = 27
SyntaxError: invalid syntax
```

Reserved words

- Check reserved words (keywords)
  – help('keywords')
  – import keyword
    print(keyword.kwlist)



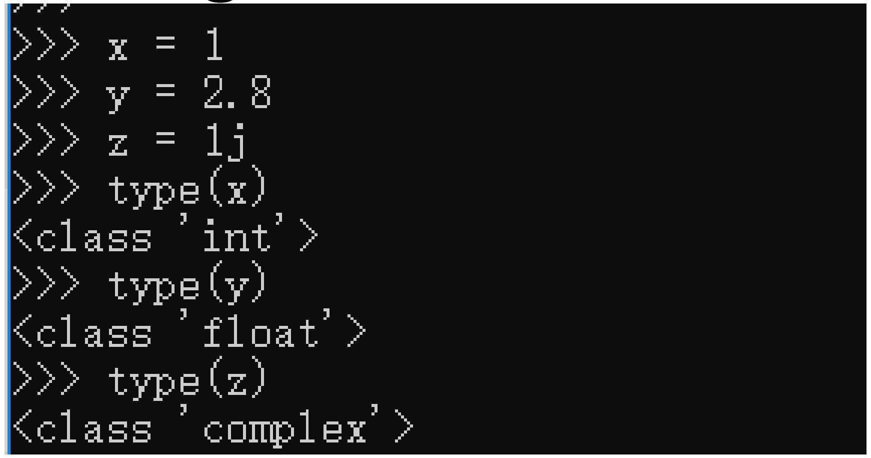
```
>>> import keyword
>>> print(keyword.kwlist)
['False', 'None', 'True', 'and', 'as', 'assert', 'async', 'await', 'break', 'class', 'continue', 'def', 'del', 'elif', '
else', 'except', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pas
s', 'raise', 'return', 'try', 'while', 'with', 'yield']
>>> help('keywords')

Here is a list of the Python keywords.  Enter any keyword to get more help.

False               class               from                or
None                continue            global              pass
True                def                 if                  raise
and                 del                 import              return
as                  elif                in                  try
assert              else                is                  while
async               except              lambda              with
await               finally             nonlocal            yield
break               for                 not
```

Variable type

- The programmer (and the interpreter) can identify the type of a variable.
- You do not need to explicitly define or declare the type of a variable.
  - ❑ int x = 3 (for most other languages)
  - ❑ x = 3 (for Python)
- Python's data types
  - ❑ Numbers
  - ❑ String
  - ❑ List
  - ❑ Tuple
  - ❑ Dictionary
  - ❑ Bool

Numbers

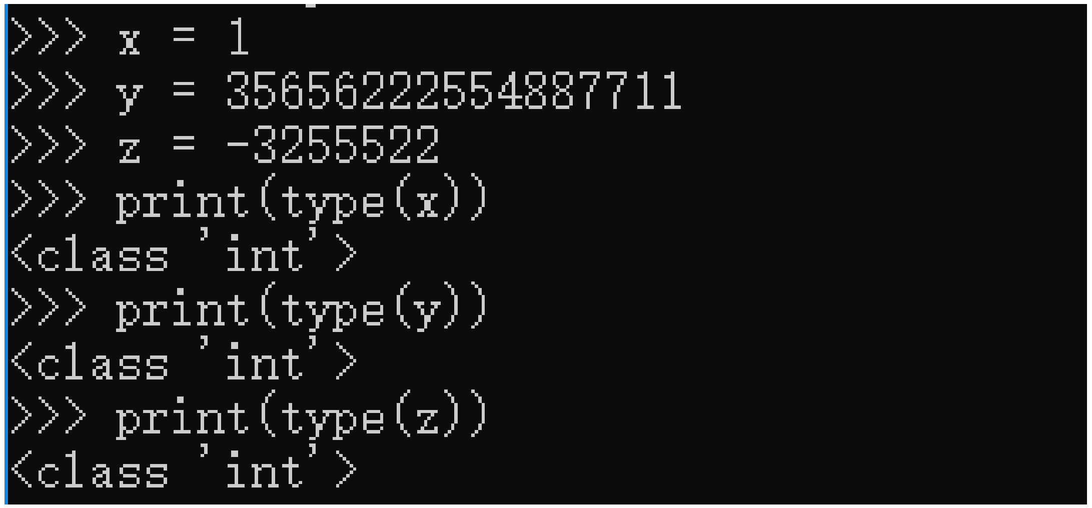
- Python 3.X supports three numerical types:
  - ❑int (signed integers)
  - ❑float (floating point real values)
  - ❑complex (complex numbers)
- Variables of numeric types are created when you assign a value to them:



```
>>> x = 1
>>> y = 2.8
>>> z = 1j
>>> type(x)
<class 'int'>
>>> type(y)
<class 'float'>
>>> type(z)
<class 'complex'>
```

Int

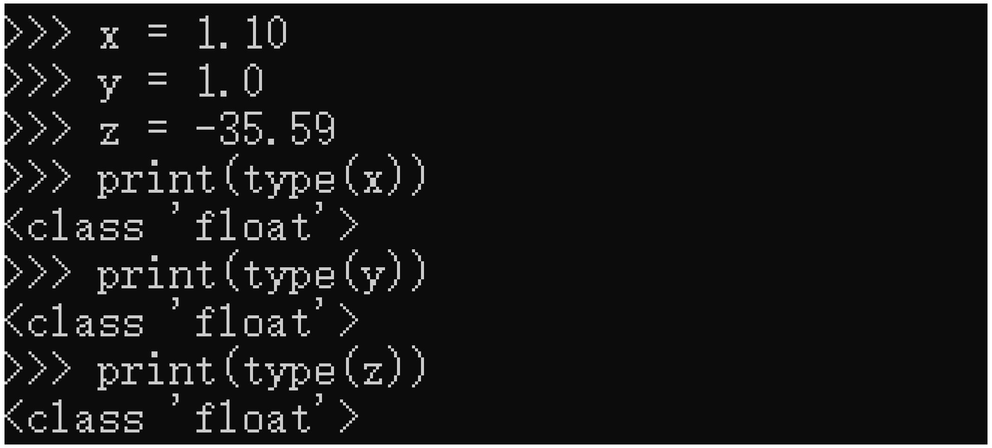
- Int, or integer, is positive or negative, without decimals, of unlimited length.



```
>>> x = 1
>>> y = 35656222554887711
>>> z = -3255522
>>> print(type(x))
<class 'int'>
>>> print(type(y))
<class 'int'>
>>> print(type(z))
<class 'int'>
```

Float

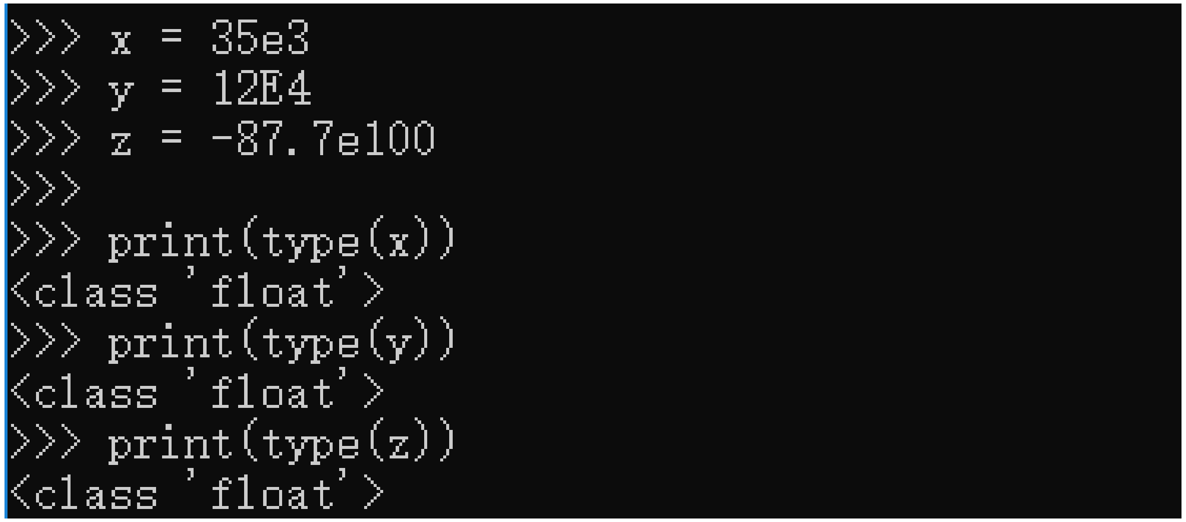
- Float, or "floating point number" is a number, positive or negative, containing one or more decimals.



```
>>> x = 1.10
>>> y = 1.0
>>> z = -35.59
>>> print(type(x))
<class 'float'>
>>> print(type(y))
<class 'float'>
>>> print(type(z))
<class 'float'>
```

Float

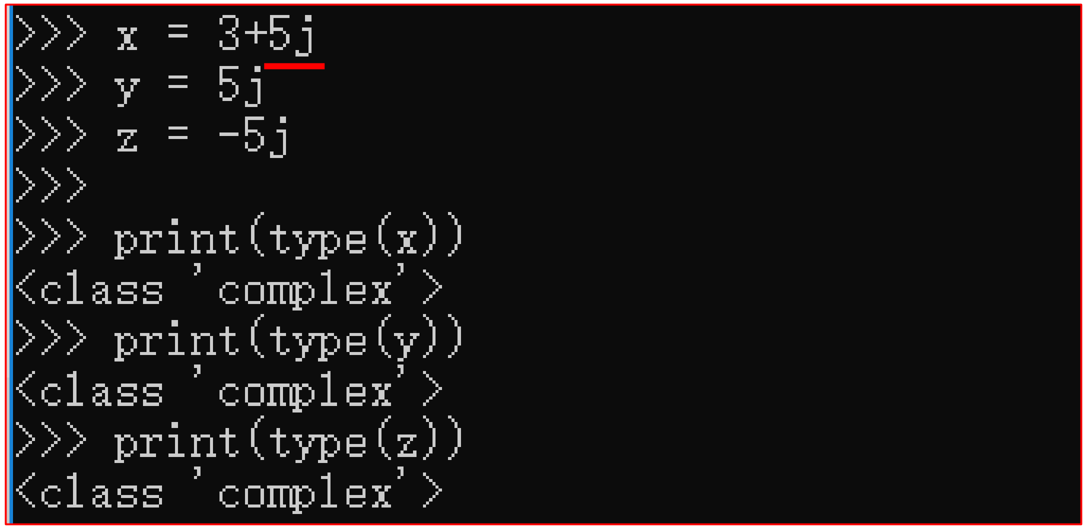
- Float can also be scientific numbers with an "e" to indicate the power of 10.



```
>>> x = 35e3
>>> y = 12E4
>>> z = -87.7e100
>>>
>>> print(type(x))
<class 'float'>
>>> print(type(y))
<class 'float'>
>>> print(type(z))
<class 'float'>
```

Complex

- Complex numbers are written with a "j" as the imaginary part:



```
>>> x = 3+5j
>>> y = 5j
>>> z = -5j
>>>
>>> print(type(x))
<class 'complex'>
>>> print(type(y))
<class 'complex'>
>>> print(type(z))
<class 'complex'>
```

Strings

- Strings in Python are identified as a contiguous set of characters represented in the single or double quotes.
- Subsets of strings can be taken using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the string.

```
#!/usr/bin/python

mystr = 'Hello World!'

print(mystr)            # Prints complete string
print(mystr[0])         # Prints first character of the string
print(mystr[2:5])       # Prints characters starting from 3rd to 5th
print(mystr[2:])        # Prints string starting from 3rd character
```

Output:

```
Hello World!
H
llo
llo World!
```

String	H	e	l	l	o		W	o	r	l	d	!
Index	0	1	2	3	4	5	6	7	8	9	10	11
Location	1	2	3	4	5	6	7	8	9	10	11	12

String methods

- The strip() method removes any whitespace from the beginning or the end:

```
a = " Hello, World! "
print(a.strip()) # returns "Hello, World!"
```

- The len() method returns the length of a string: len(a)
- String concatenate: print(a+'!!!!') `#  Hello, World! !!!!`
- The lower() method returns the string in lower case a.lower()
- The upper() method returns the string in upper case a.upper()
- The replace() method replaces a string with another string:

```
a = "Hello, World!"
print(a.replace("H", "J")) #Jello, World!
print(a) #but string a is not changed
```

- The split() method splits the string into substrings if it finds instances of the separator:

```
a = "Hello, World!"
print(a.split(",")) # returns ['Hello', ' World!']
print(a.split()) #default splitter " ", returns ['Hello,', 'World!']
```

https://docs.python.org/3/library/stdtypes.html#string-methods

Lists

- A list contains items separated by commas and enclosed within square brackets([]).
- The values stored in a list can be accessed using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the list.

```
Index:      0          1       2      3       4
```

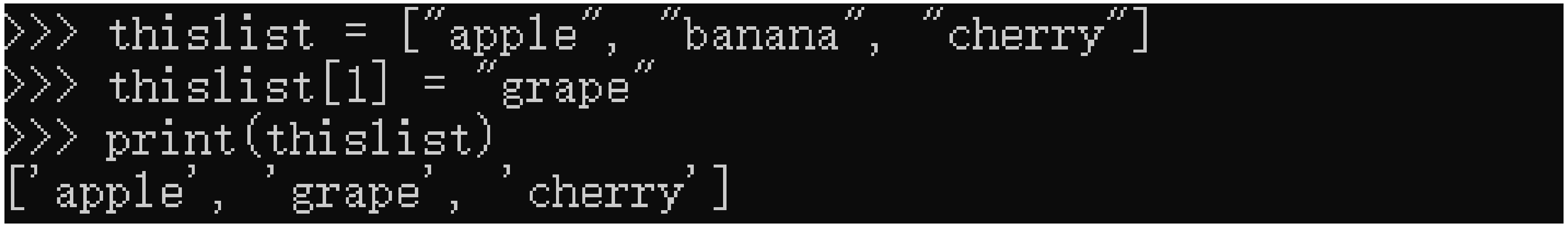
```
thislist = [ 'abcd', 786 , 2.23, 'john', 70.2 ]
tinylist = [123, 'john']

print(thislist)          # Prints complete list
print(thislist[0])       # Prints first element of the list
print(thislist[1:3])     # Prints elements starting from 2nd till 3rd
print(thislist[2:])      # Prints elements starting from 3rd element
```

Output:

```
['abcd', 786, 2.23, 'john', 70.2]
abcd
[786, 2.23]
[2.23, 'john', 70.2]
```

Lists

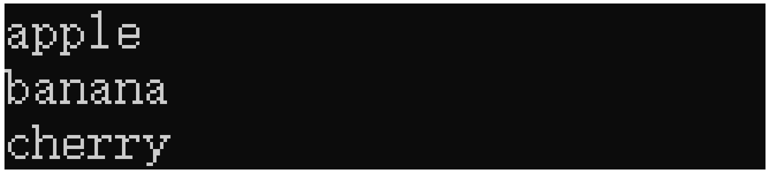
- To change the value of a specific item, refer to the index number:



```
>>> thislist = [“apple”, “banana”, “cherry”]
>>> thislist[1] = “grape”
>>> print(thislist)
['apple', 'grape', 'cherry']
```

- loop through the list items by using a for loop:

```
thislist = ["apple", "banana", "cherry"]
for x in thislist:
  print(x)
```



```
apple
banana
cherry
```

Lists

- To determine if a specified item is present in a list use the in keyword:

```
if "apple" in thislist:
  print("Yes, 'apple' is in the fruits list")
```

- len(thislist)
- thislist.append("orange")
- thislist.insert(1, "orange")
- thislist.remove("banana")

https://docs.python.org/3/tutorial/datastructures.html#more-on-lists

Tuples

- A tuple is another sequence data type that is similar to the list.
- A tuple contains items separated by commas and enclosed within parentheses ().
- Difference between lists and tuples:
  - ❑ Elements and size in lists can be changed, while tuples can not

```
#!/usr/bin/python

thistuple = ( 'abcd', 786 , 2.23, 'john', 70.2  )
tinytuple = (123, 'john')

print(thistuple)          # Prints complete list
print(thistuple[0])       # Prints first element of the list
print(thistuple[1:3])     # Prints elements starting from 2nd till 3rd
print(thistuple[2:])      # Prints elements starting from 3rd element
```

Output:

```
('abcd', 786, 2.23, 'john', 70.2)
abcd
(786, 2.23)
(2.23, 'john', 70.2)
```

Tuples

- You cannot change values in a tuple:

```
thistuple = ("apple", "banana", "cherry")
thistuple[1] = "blackcurrant"
# The values will remain the same:
print(thistuple)
```

- Loop Through a Tuple

```
thistuple = ("apple", "banana", "cherry")
for x in thistuple:
  print(x)
```

- Check if Item Exists

```
thistuple = ("apple", "banana", "cherry")
if "apple" in thistuple:
  print("Yes, 'apple' is in the fruits tuple")
```

Dictionary

- A dictionary is a collection which is unordered, changeable and indexed.
- Consists of a number of key-value pairs.
- It is enclosed by curly braces ({ })

```
thisdict={"brand": "Ford","model": "Mustang","year": 1964}
print(thisdict)
```

Dictionary

- Get value

```
x = thisdict["model"]
```

- Change Values

```
thisdict["year"] = 2018
```

- Loop through keys

```
for x in thisdict:
  print(x)
```

- Loop through values

```
for x in thisdict.values():
  print(x)
```

- Add/delete items

```
thisdict["color"] = "red"
del thisdict["model"]
```

Set

- A collection which is unordered and unindexed, written with curly brackets.

```
{1,2,'a','b'}
set() # empty set, not  {}
```

- **Common uses include**
  - membership testing
  - removing duplicates from a sequence
  - intersection, union, difference, and symmetric difference

Set

```
>>> a = {3, 'a', 2, 1,11,15,'1','2'}
>>> a.pop()
1
>>> a.remove(11)
>>> a
{2, 3, '2', '1', 15, 'a'}
>>> a.add(20)
>>> a
{2, 3, '2', '1', 15, 20, 'a'}
>>> b = {2,3}
>>> b.issubset(a)
True
>>> c = {5,6}
>>> c.isdisjoint(a)
True
>>> x = {1,1.0,1.00}
>>> x
{1}
```

pop: remove and return an arbitrary set element

pop and remove raise KeyError if the set does not contain the element

Elements are compared via ==

Set

```
>>> a_set = set([8, 9, 10, 11, 12, 13])
>>> b_set = {0, 1, 2, 3, 7, 8}
>>> a_set | b_set                     # union
{0, 1, 2, 3, 7, 8, 9, 10, 11, 12, 13}
>>> a_set.union(b_set)                # union
{0, 1, 2, 3, 7, 8, 9, 10, 11, 12, 13}
>>> a_set & b_set                # intersection
{8}
>>> a_set.intersection(b_set)    # intersection
{8}
>>> a_set.difference(b_set)      # difference
{9, 10, 11, 12, 13}
>>> a_set - b_set
{9, 10, 11, 12, 13}
```

Bool type

- Only has two values: True and False
- True equals to int 1

```
True == 1
```

- Any zero numerical values are False

```
False == 0
False == 0.0
False == 0j
```

- Associated with Logic and Comparison Operations

Get variable type

- If you are not sure what type a variable has, the interpreter can tell you by using **type**.
  - ❑>>> type('Hello, world!')

    <type 'str'>
  - ❑>>> type(17)

    <type 'int'>
  - ❑>>> type(3.2)

    <type 'float'>
  - ❑>>> type('4.7')

    <type 'str'>

Type matters

- Python knows what “type” everything is.
- Some operations are prohibited.
  - ❑You can not add a string to an integer.

```
>>> a = '123'
>>> b = a + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and
'int' objects
```

Type matters

- Operations on the same type operands will lead to results with the same type, except division
  - ❑int + int => int
  - ❑int – int => int
  - ❑int * int => int
  - ❑int / int => float   (if in Python2, int)

Type conversions

- Covert type with the built-in functions int(), float(), str()
  - int() - constructs an integer number from an integer literal, a float literal (by rounding down to the previous whole number), or a string literal (providing the string represents a whole number)
  - float() - constructs a float number from an integer literal, a float literal or a string literal (providing the string represents a float or an integer)
  - str() - constructs a string from a wide variety of data types, including strings, integer literals and float literals

Type conversions

```
x = int(1)   # x will be 1
y = int(2.8) # y will be 2
z = int("3") # z will be 3
```

```
x = float(1)     # x will be 1.0
y = float(2.8)   # y will be 2.8
z = float("3")   # z will be 3.0
w = float("4.2") # w will be 4.2
```

```
x = str("s1") # x will be 's1'
y = str(2)    # y will be '2'
z = str(3.0)  # z will be '3.0'
```

User input

- We can ask Python to pause and read data from the keyboard using input() function.
- The input() function returns a string.

```
>>> name = input("What's your name? ")
What's your name? HP # You type using keyboard
>>> print('Welcome', name)
Welcome HP
```

- If we want to read a number from the keyboard, we must convert it from a string to a number using a type conversion function [int() or float()].

```
>>> inp = input('Europe floor? ')
Europe floor? 2 # 2 is the input from keyboard
>>> usf = int(inp) + 1
>>> print('US floor', usf)
US floor 3
```

The assignment statement

- A **statement** is an instruction that the Python interpreter can execute. E.g. if statement, while statement, assignment statement…
- An assignment statement creates new variables and gives them values.
- An assignment statement consists of an expression on the right-hand side and a variable to store the result.
  - >>> message = “hello, world”
  - >>> x = 17
  - >>> pi = 3.14159

Expressions

- An expression is a combination of values, variables, and operators. Not every expression contains all of these elements.
- If you type an expression on the command line (after >>>), the interpreter **evaluates** it and displays the result.

```
❑>>> 1+2*7
      15
❑>>>'Hello'
      Hello
```

Operators

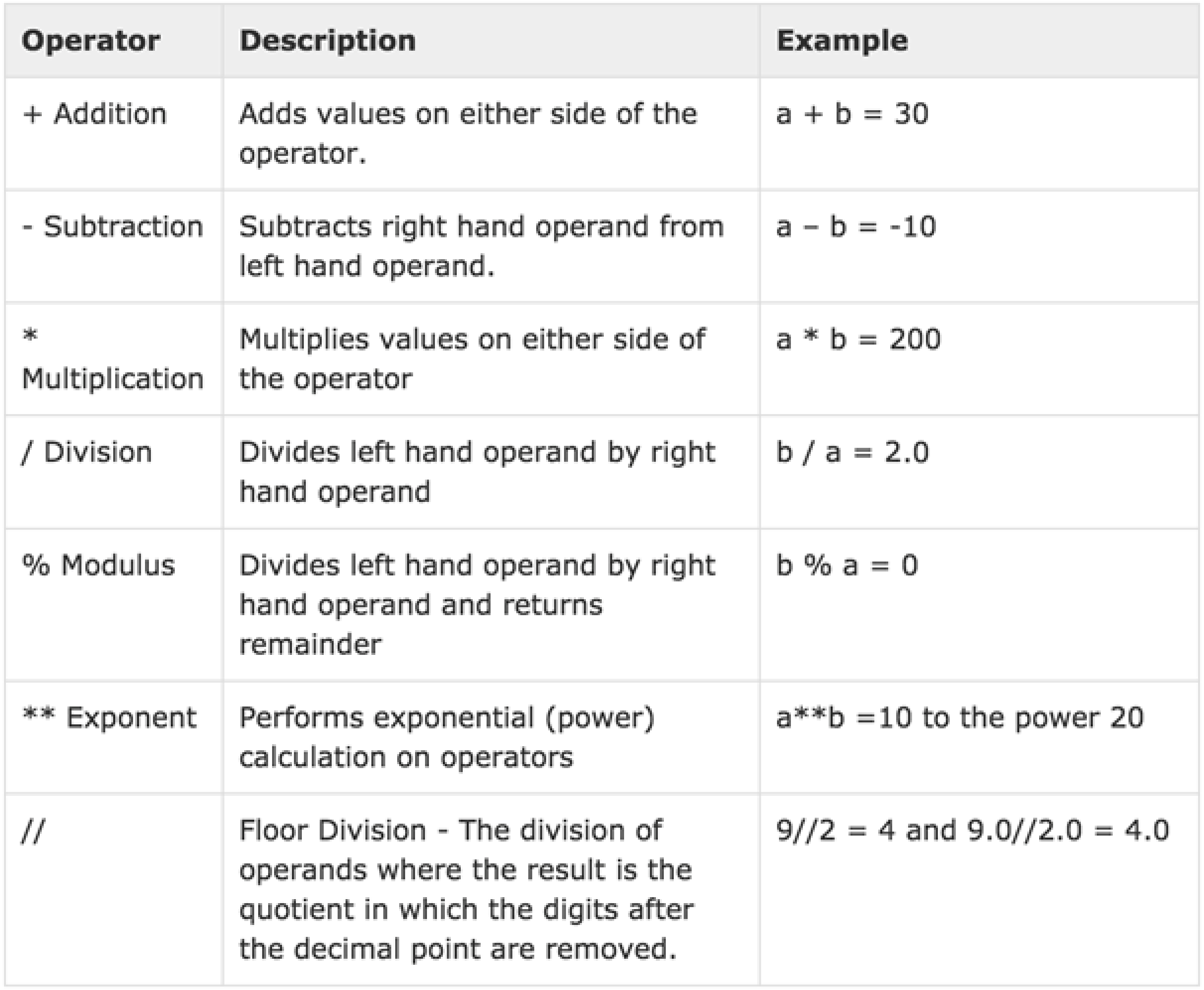
- Operators are special symbols that represent computations like addition and multiplication.
- The values the operator uses are called operands.
- Types of operator
    - ❑ Arithmetic operators
    - ❑ Comparison (Relational) operators
    - ❑ Assignment operators
    - ❑ Logic operators
    - ❑ Bitwise operators
    - ❑ Membership operators
    - ❑ Identity operators

Arithmetic Operators

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

a = 10

b = 20



Operator	Description	Example
+ Addition	Adds values on either side of the operator.	a + b = 30
- Subtraction	Subtracts right hand operand from left hand operand.	a – b = -10
* Multiplication	Multiplies values on either side of the operator	a * b = 200
/ Division	Divides left hand operand by right hand operand	b / a = 2.0
% Modulus	Divides left hand operand by right hand operand and returns remainder	b % a = 0
** Exponent	Performs exponential (power) calculation on operators	a**b =10 to the power 20
//	Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed.	9//2 = 4 and 9.0//2.0 = 4.0

The modulus operator

- It works on integers (and integer expressions) and yields the remainder when the first operand is divided by the second.

```
>>> remainder = 7 % 3
>>> print(remainder)
1
```

Use case (1)

- You can check whether one number is divisible by another – if x % y is zero, the x is divisible by y

```
>>> is_divisible = 6 % 3
>>> print(is_divisible)
0
>>> is_not_divisible = 7 % 3
>>> print(is_not_divisible)
1
```

Use case (2)

- ❑You can extract the right-most digit or digits from a number.
  - ❑For example, the last two digits in 123491 is 91.

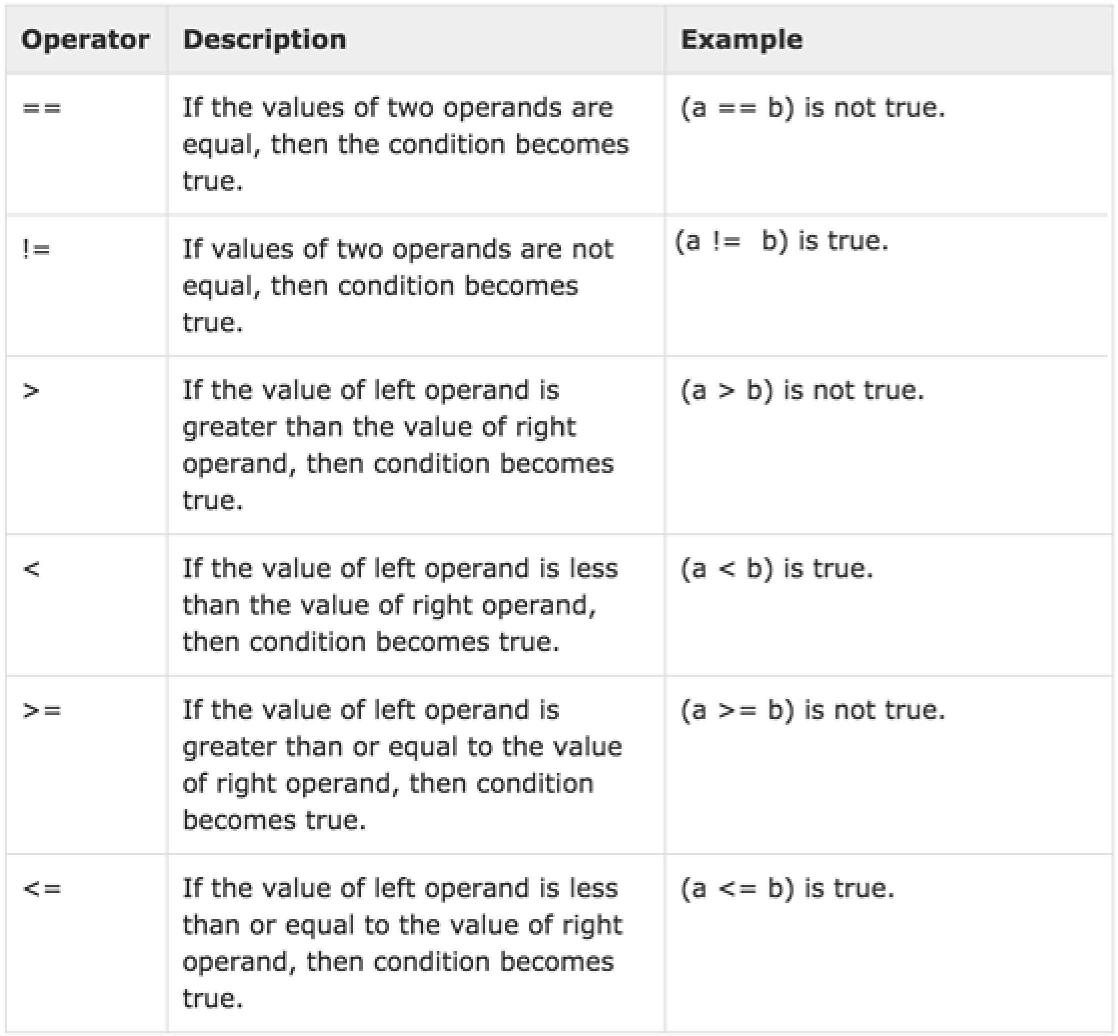
```
>>> last_two_digits = 123491 % 100
>>> print(last_two_digits)
91
>>> last_one_digit = 123491 % 10
>>> print(last_one_digit)
1
```

Comparison operators

- Compare values on either sides of them and decide the relation among them. **Give bool type results.**

a = 10

b = 20



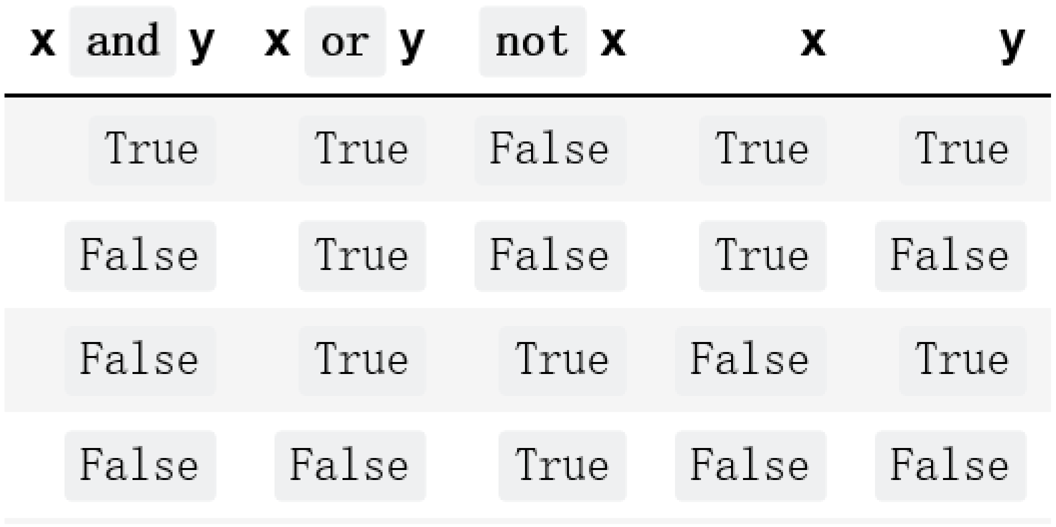
Operator	Description	Example
==	If the values of two operands are equal, then the condition becomes true.	(a == b) is not true.
!=	If values of two operands are not equal, then condition becomes true.	(a != b) is true.
>	If the value of left operand is greater than the value of right operand, then condition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

Comparison: Example

```
>>> 1 == 1.0
True
>>> 1 > 2.0
False
>>> 1 != 1.0
False
>>> 1 >= 1.0
True
```

Logic operators

- Performs logic operations: and, or, not



x and y	x or y	not x	x	y
True	True	False	True	True
False	True	False	True	False
False	True	True	False	True
False	False	True	False	False

Logical operators ascending priority

or and not $\longrightarrow$

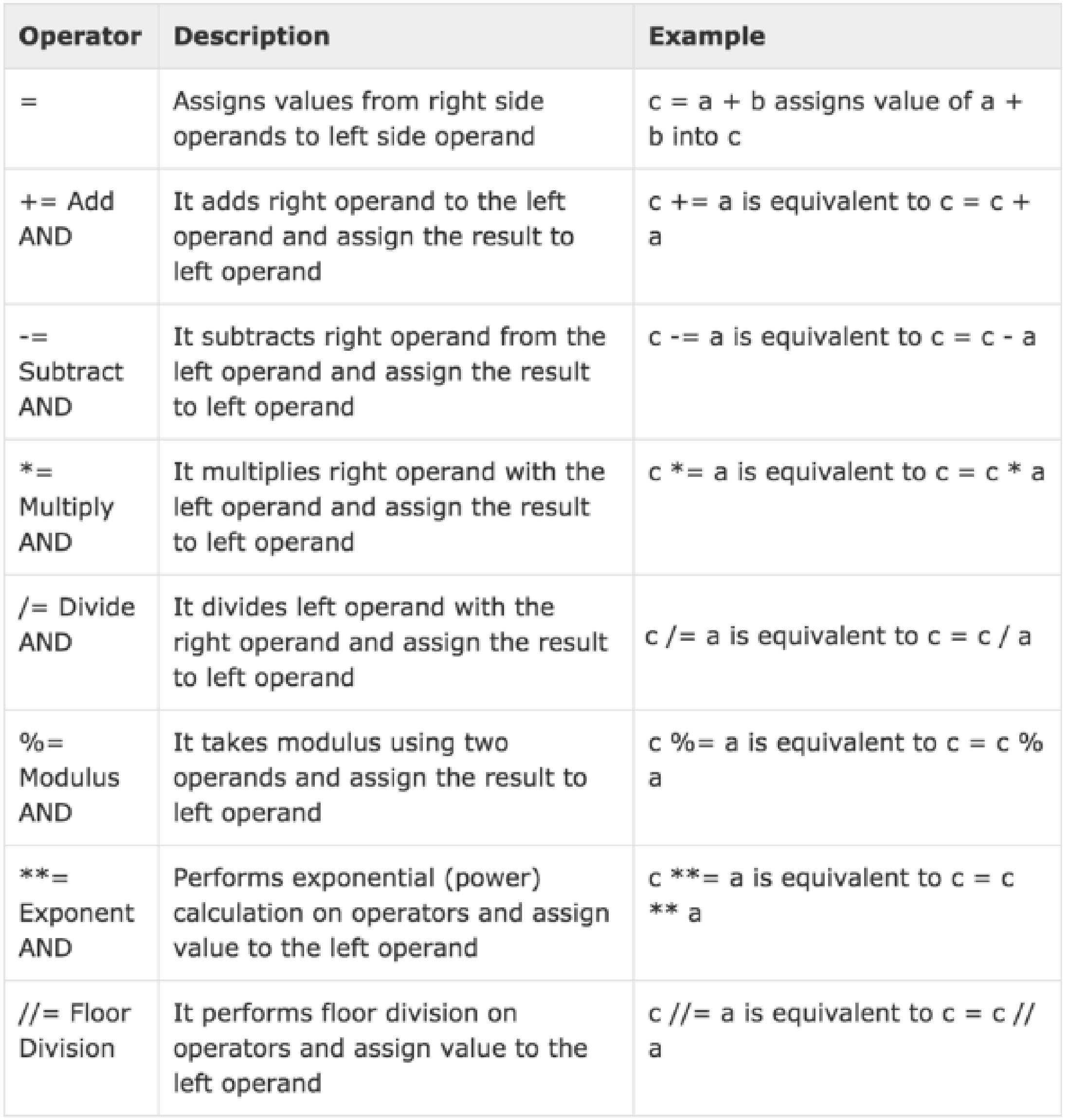
Boolean: Example

```python
print(True or True and False)
print(True or (True and False))
print((True or True) and False)
```

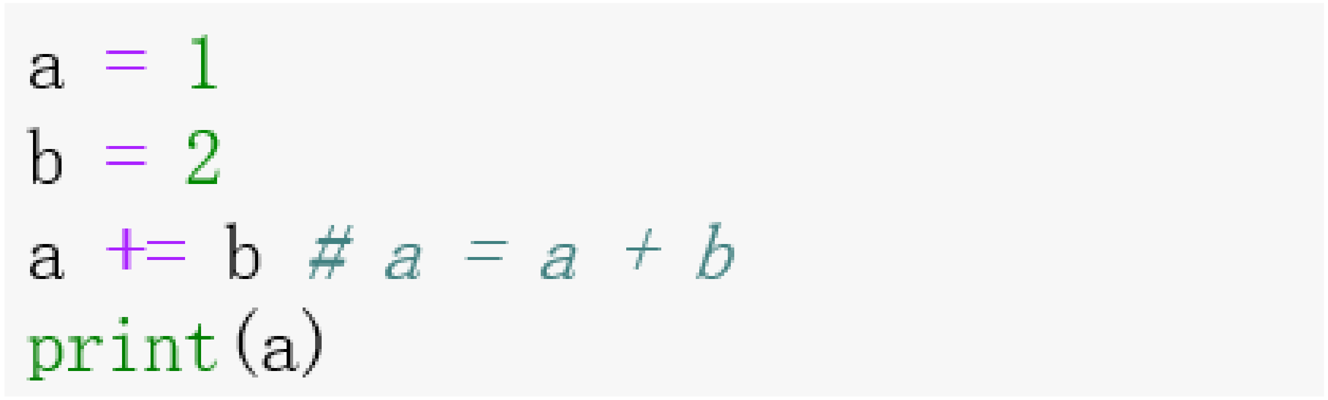
Output

```
True
True
False
```

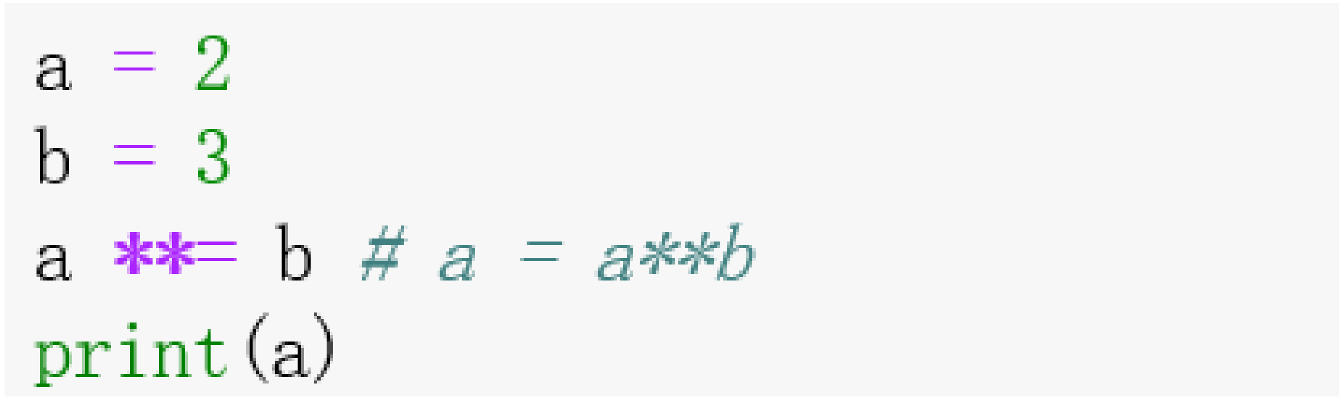
Assignment operators



Operator	Description	Example
=	Assigns values from right side operands to left side operand	c = a + b assigns value of a + b into c
+= Add AND	It adds right operand to the left operand and assign the result to left operand	c += a is equivalent to c = c + a
-= Subtract AND	It subtracts right operand from the left operand and assign the result to left operand	c -= a is equivalent to c = c - a
*= Multiply AND	It multiplies right operand with the left operand and assign the result to left operand	c *= a is equivalent to c = c * a
/= Divide AND	It divides left operand with the right operand and assign the result to left operand	c /= a is equivalent to c = c / a
%= Modulus AND	It takes modulus using two operands and assign the result to left operand	c %= a is equivalent to c = c % a
**= Exponent AND	Performs exponential (power) calculation on operators and assign value to the left operand	c **= a is equivalent to c = c ** a
//= Floor Division	It performs floor division on operators and assign value to the left operand	c //= a is equivalent to c = c // a



```
a = 1
b = 2
a += b # a = a + b
print(a)
```



```
a = 2
b = 3
a **= b # a = a**b
print(a)
```

Bitwise operators

Assume variable a holds 60
and variable b holds 13

Binary representation:

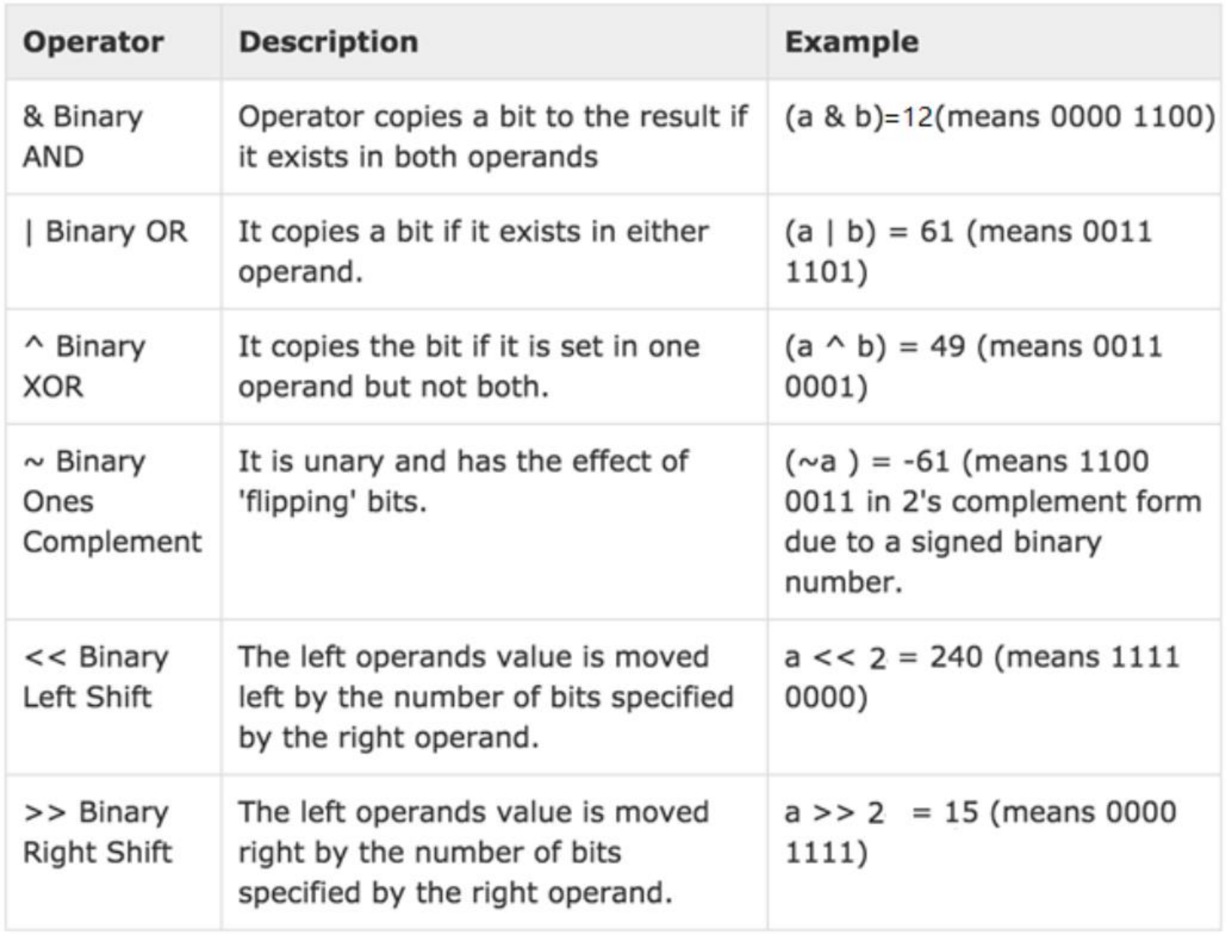
a = 0011 1100

b = 0000 1101

a & b = 0000 1100

a | b = 0011 1101

a ^ b = 0011 0001



Operator	Description	Example
& Binary AND	Operator copies a bit to the result if it exists in both operands	(a & b)=12(means 0000 1100)
| Binary OR	It copies a bit if it exists in either operand.	(a | b) = 61 (means 0011 1101)
^ Binary XOR	It copies the bit if it is set in one operand but not both.	(a ^ b) = 49 (means 0011 0001)
~ Binary Ones Complement	It is unary and has the effect of 'flipping' bits.	(~a) = -61 (means 1100 0011 in 2's complement form due to a signed binary number.
<< Binary Left Shift	The left operands value is moved left by the number of bits specified by the right operand.	a << 2 = 240 (means 1111 0000)
>> Binary Right Shift	The left operands value is moved right by the number of bits specified by the right operand.	a >> 2 = 15 (means 0000 1111)

```
a = 60 #Binary representation: 0011 1100 (suppose we use 8 bits)
b = 13 #Binary representation: 0000 1101 (suppose we use 8 bits)
```

```
a<<2 #1111 0000
```

```
240
```

```
a>>2 #0000 1111
```

```
15
```

```
a & b #0000 1100
```

```
12
```

```
a | b #0011 1101
```

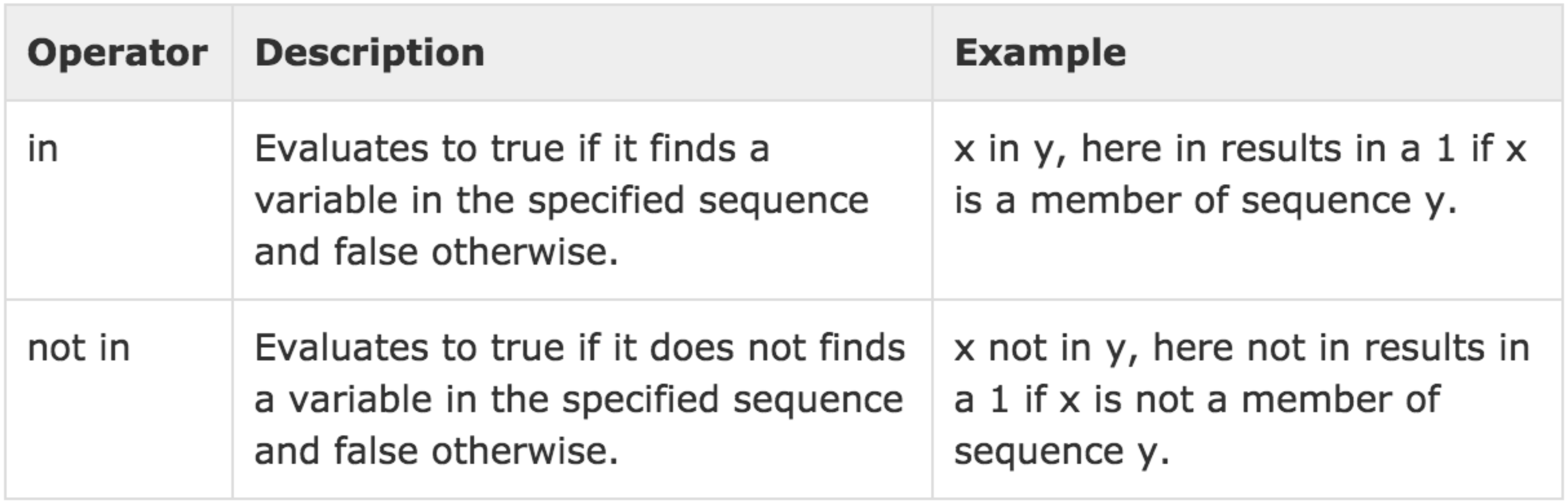
```
61
```

```
a ^ b  #0011 0001
```

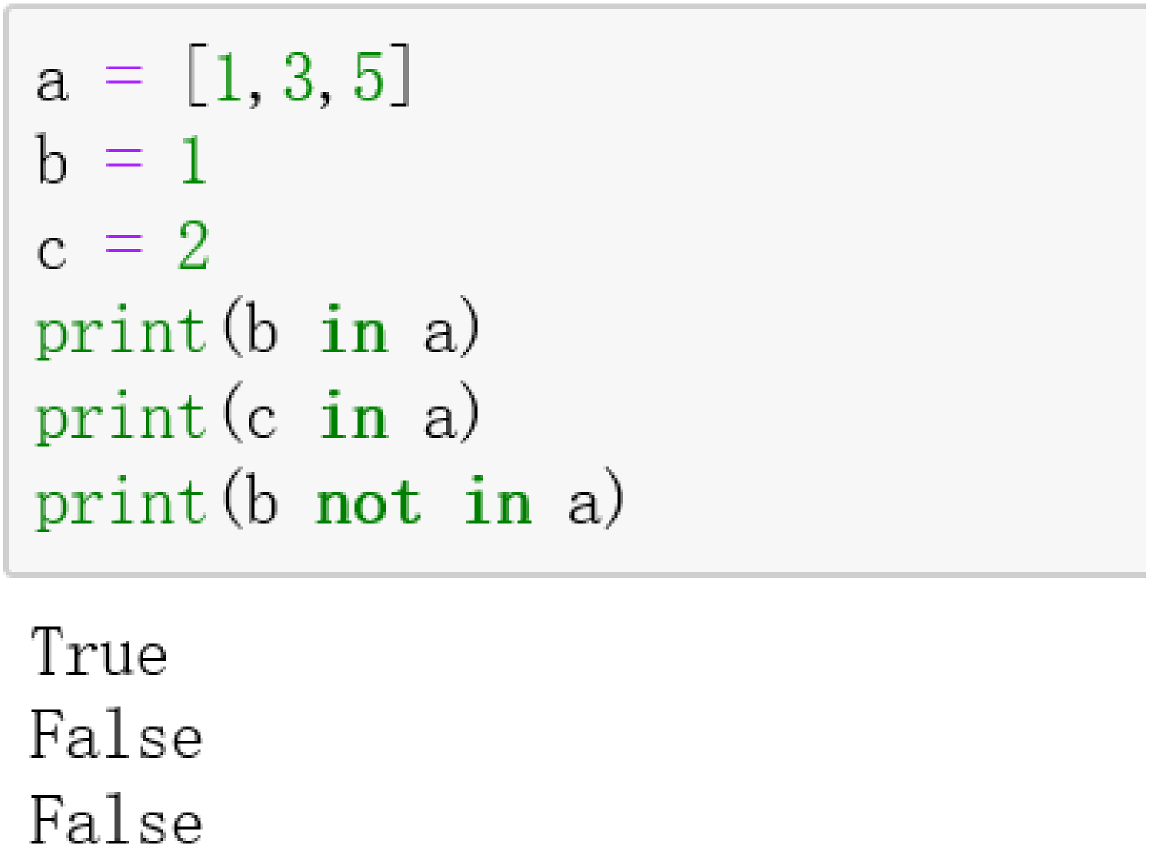
```
49
```

Membership operators

- Test for memberships in a sequence, such as strings, lists, or tuples.



Operator	Description	Example
in	Evaluates to true if it finds a variable in the specified sequence and false otherwise.	x in y, here in 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.

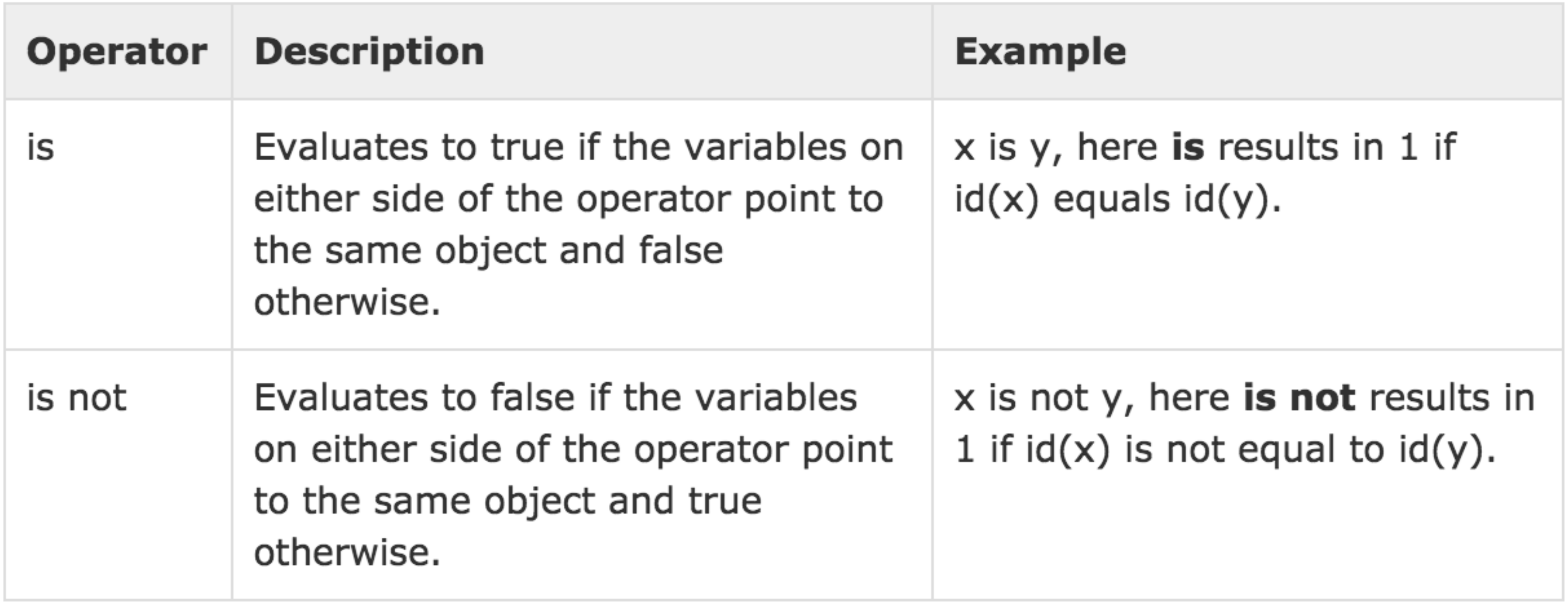


```python
a = [1,3,5]
b = 1
c = 2
print(b in a)
print(c in a)
print(b not in a)
```

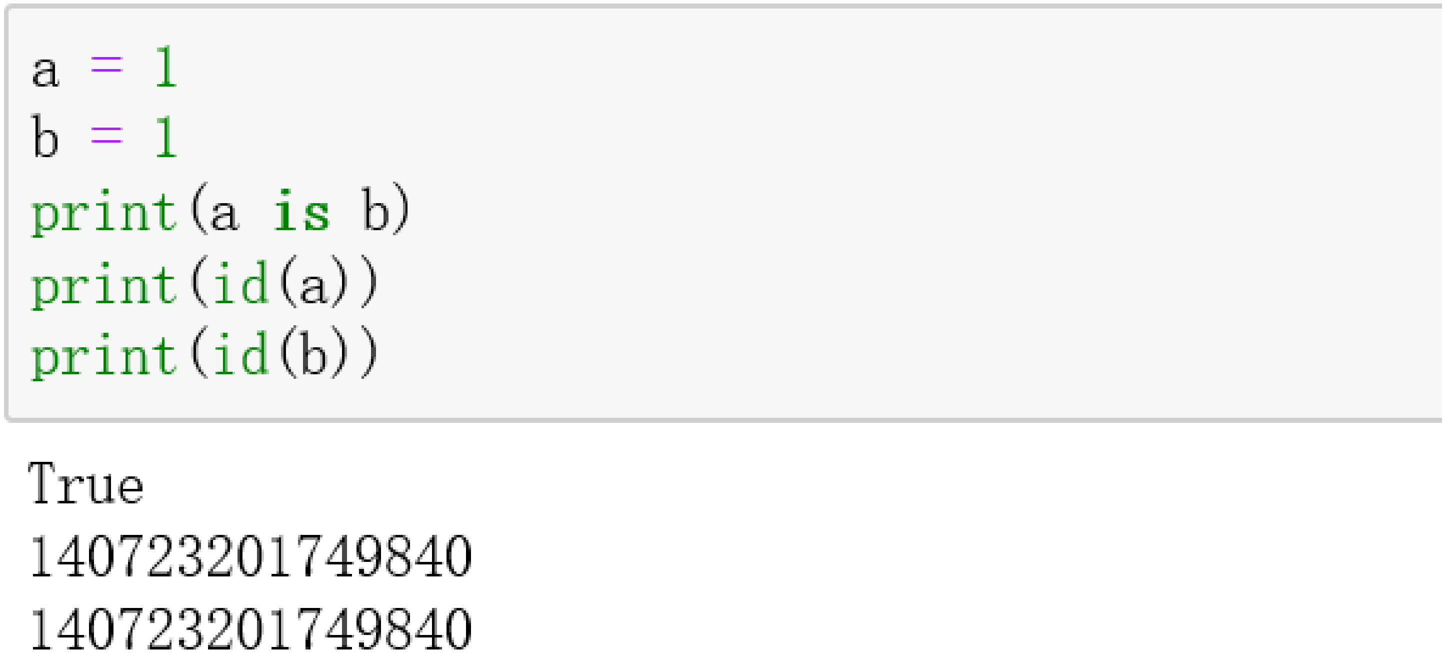
```
True
False
False
```

Identity operators

Compare the memory location of two objects.



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).



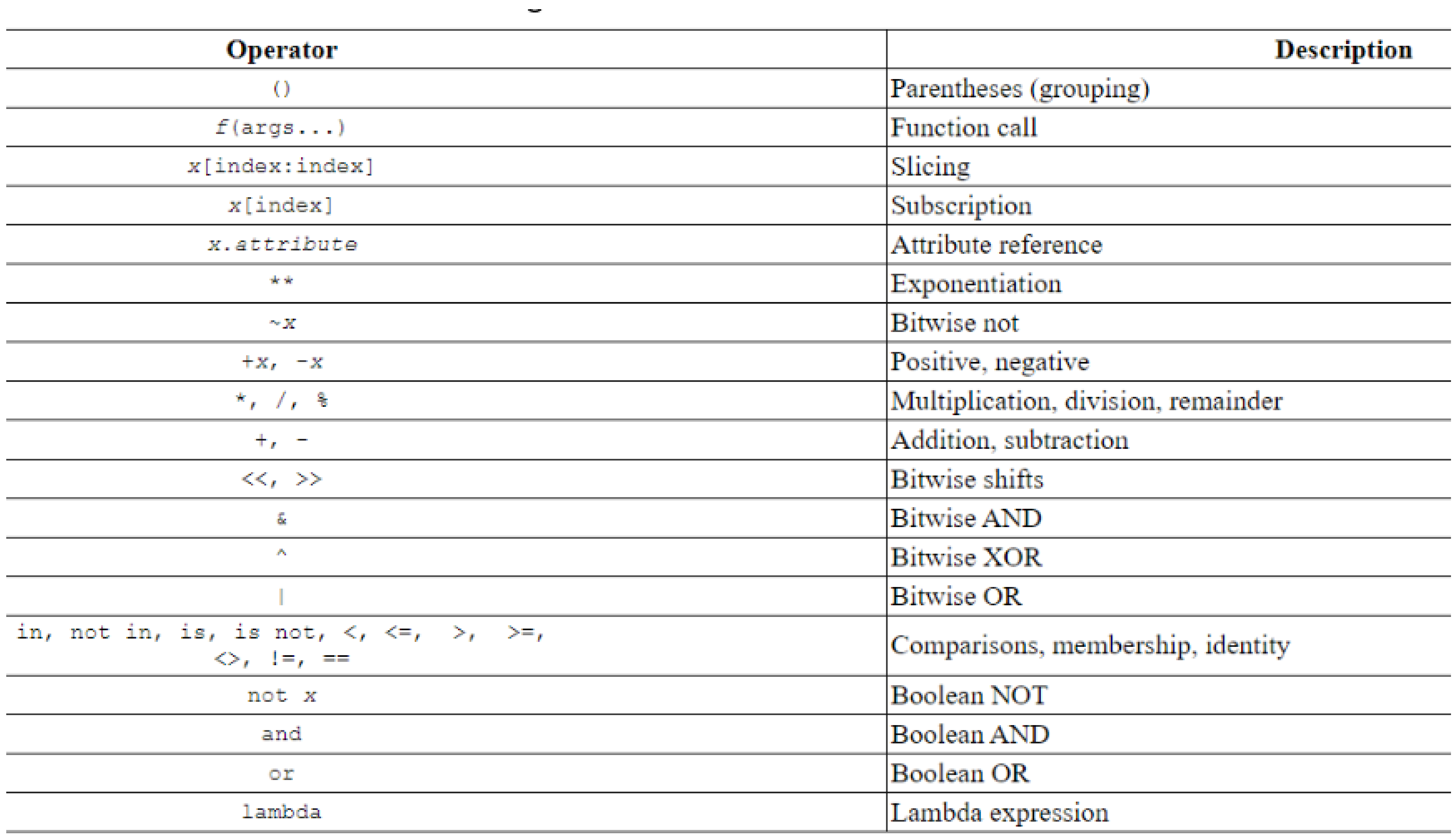
```python
a = 1
b = 1
print(a is b)
print(id(a))
print(id(b))
```

```
True
140723201749840
140723201749840
```

Operator precedence

- When more than one operator appears in an expression, the order of evaluation depends on the rules of precedence.
  - ❑Parentheses have the highest precedence and can be used to force an expression to evaluate in the order you want.
  - ❑Exponentiation has the next highest precedence.
  - ❑Multiplication and Division have the same precedence, which is higher than Addition and Subtraction, which also have the same precedence
  - ❑Operators with the same precedence are evaluated from left to right.

Operator precedence



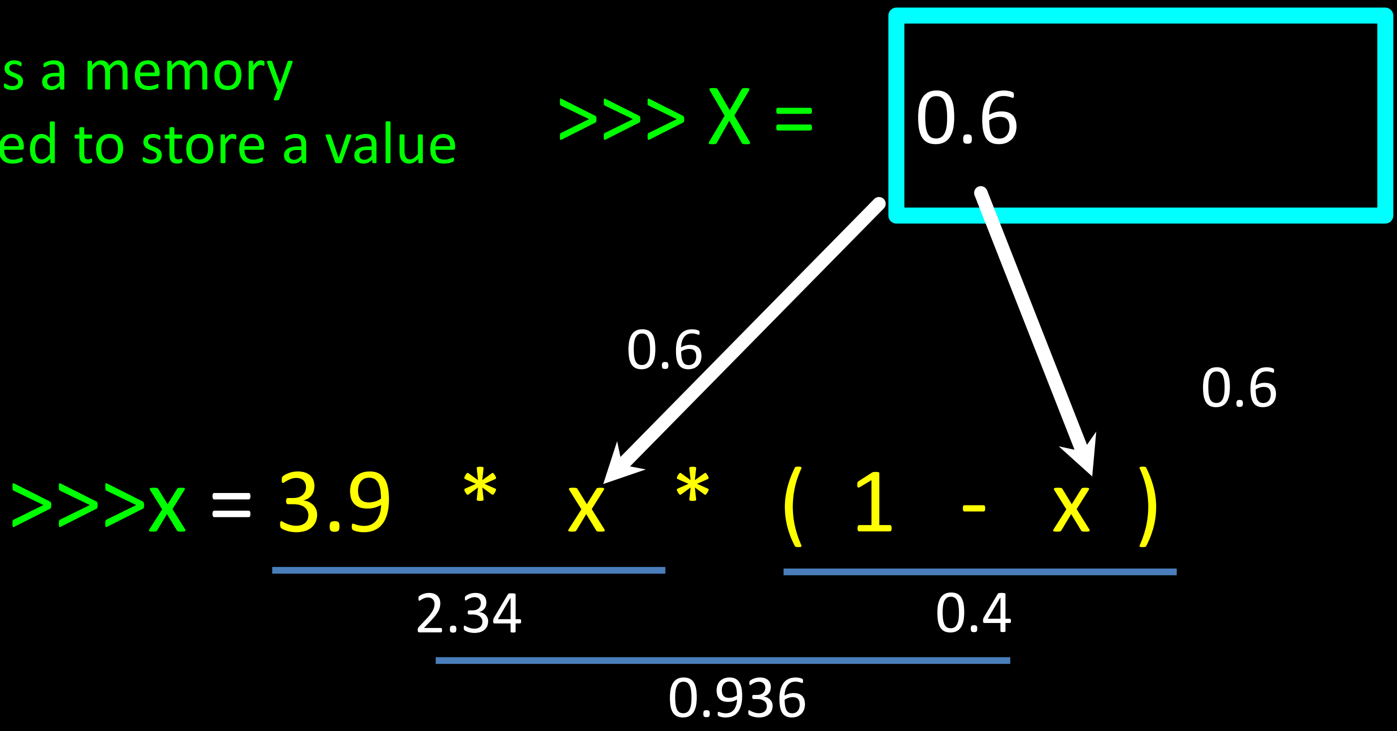
Operator	Description
`()`	Parentheses (grouping)
`f(args...)`	Function call
`x[index:index]`	Slicing
`x[index]`	Subscription
`x.attribute`	Attribute reference
`**`	Exponentiation
`~x`	Bitwise not
`+x, -x`	Positive, negative
`*, /, %`	Multiplication, division, remainder
`+, -`	Addition, subtraction
`<<, >>`	Bitwise shifts
`&`	Bitwise AND
`^`	Bitwise XOR
`|`	Bitwise OR
`in, not in, is, is not, <, <=, >, >=, <>, !=, ==`	Comparisons, membership, identity
`not x`	Boolean NOT
`and`	Boolean AND
`or`	Boolean OR
`lambda`	Lambda expression

Assignment statement

- An assignment statement consists of an expression on the right-hand side and a variable to store the result.

```
X = 3.9 * X * (1 - X)
```

A variable is a memory location used to store a value (0.6)



```
>>> X = 0.6
>>>x = 3.9 * x * ( 1 - x )
```

The right side is an expression.
Once the expression is evaluated, the result is placed in (assigned to) x.

A variable is a memory location used to store a value. The value stored in a variable can be updated by replacing the old value (0.6) with a new value (0.93).

$$X = \boxed{\cancel{0.6} \quad 0.936}$$

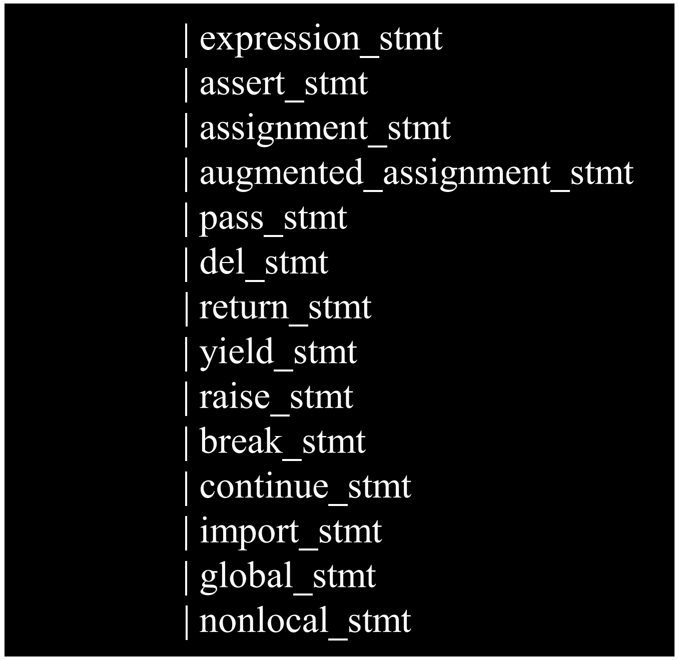
```
x = 3.9  *  x  *  ( 1  -  x )
```

0.936

The right side is an expression. Once the expression is evaluated, the result is placed in (assigned to) the variable on the left side (i.e., x).

Statements

- Simple statements: executed sequentially and do not affect the flow of control.
  - ❑ Print statement
  - ❑ Assignment statement
  - ❑ And many others…



```
| expression_stmt
| assert_stmt
| assignment_stmt
| augmented_assignment_stmt
| pass_stmt
| del_stmt
| return_stmt
| yield_stmt
| raise_stmt
| break_stmt
| continue_stmt
| import_stmt
| global_stmt
| nonlocal_stmt
```

- Compound statements: may affect the sequence of execution.

Readings (recommended)

- The Python Tutorial
  - 4.1 to 4.6 in 4. More Control Flow Tools

Recap

- Variables
- Expressions
- Statements