# Python Lecture 2 – Programming

- Data types
- Lambda expressions
- Input
- Flow control instructions
- Iterators
- Simple matrix examples

S. Bertocco

# Bibliography and learning materials

- ★ Bibliography:

  https://www.python.org/doc/

  http://docs.python.it/

  and much more available in internet

- ★ Learning Materials:

  https://github.com/gtaffoni/Learn-Python/tree/master/Lectures

  https://github.com/bertocco/bash_lectures

# Data types

Python has two families of data types:

**Simple data** types:

– Int

– Float

– Complex

– Boolean

– String

**Container** data types:

– list []

– tuple ()

– dict {}

– set

# Simple Data Types

- – Int
- – Float
- – Complex
- – Boolean
- – String

# Numeric types

In python there are 3 numeric types:

- Integer
- Float
- Complex
- + Boolean (extension of int)

In general, an n-bit integer has values ranging
from -2^(n-1) to 2^(n-1) – 1

information about integer dimension:

```
>>> sys.maxsize
9223372036854775807
```

information about the internal representation of floating point

```
>>> sys.float_info
sys.float_info(max=1.7976931348623157e+308, max_exp=1024,
max_10_exp=308, min=2.2250738585072014e-308, min_exp=-1021,
min_10_exp=-307, dig=15, mant_dig=53,
        epsilon=2.220446049250313e-16, radix=2, rounds=1)
```

# Numeric types: Examples

int type can be:

in base 2 (using the prefix 0b) bin(19)
in base 10,
in base 16 (using the prefix 0x) hex(300)
in base 8 (using the prefix 0) oct(300)

```
>>> a=300
>>> oct(a)
'0454'
>>> hex(a)
'0x12c'
>>> bin(a)
'0b100101100'
>>> bin(19)
'0b10011'
```

# Numeric types

float are real number in double precision.

Examples:

```
>>> a = 12.456
>>> c = 12232e-2
>>> b = .2
>>> a=6.12244e-5
>>> type(a)
<class 'float'>
```

**Note**: floor (troncamento) function is the function that takes as input a real number x and gives as output the greatest integer less than or equal to x.

floor: 2.1 → 2 -0.1 → -1

The ceiling (arrotondamento) function maps x to the least integer greater than or equal to x

ceiling: -0.99 → 0 2.1 → 3

Be careful using int and float:

What happens doing...

| | |
|---|---|
| 100/3 | division int/int |
| 100//3 | floor division int/int (gets the integer part) |
| 100.0/3 | division float/int |
| 100.0//3 | floor division float/int |
| 100%3 | remainder of the division int/int |
| divmod(100,3) | The divmod() method takes two numbers and returns a pair of numbers (a tuple) consisting of their quotient and remainder. |

# Numeric types

Complex Number represents a complex number in double precision. The real and the imaginary parts can be accessed using the functions ‘real’ and ‘imag’.

**Example**:

```
>>> r=12+5j              # ‘j’ symbol means the imaginary part
>> r=10+5j
>>> type(r)
<class 'complex'>

>>> r.real
12.0
>>> r.imag
5.0
>>> type(r.real)
<type 'float'>
>>> type(r.imag)
<type 'float'>
```

# Operations on numeric types

In Python operations on numeric types are managed by the following operators:

- Arithmetic operators
- Comparison operators
- logical operators
- bitwise (bit a bit) operators
- membership operators
- identity operators

Some built-in functions working on numeric data:

- abs(number)   returns the absolute value of a number
- pow(x, y[, z])   returns the value of x to the power of y ($x^y$). If a third parameter is present, it returns x to the power of y, modulus z.
- round(number[, ndigits])   returns a floating point number that is a rounded version of the specified number, with the specified number of decimals.

Executing operations between different numeric type variables,
the implicit conversion rule is:

Int → Float → Complex

# Arithmetic operators

a=10    b=21

| Operator | Description | Example |
|---|---|---|
| + Addition | Adds values on either side of the operator. | a+b=31 |
| - Subtraction | Subtracts right hand operand from left hand operand. | a-b=-11 |
| * Multiplication | Multiplies values on either side of the operator | a*b=210 |
| / Division | Divides left hand operand by right hand operand | b/a=2.1 |
| % Modulus | Divides left hand operand by right hand operand and returns remainder | b%a=1 |
| ** Exponent | Performs exponential (power) calculation on operators | a**b=$10^{20}$ |
| // Floor division | The division of operands where the result is the quotient in which 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<br>9.0//2.0 = 4.0,<br>-11//3 = -4,<br>-11.0//3 = -4.0 |

https://www.tutorialspoint.com/python3/python_basic_operators.htm

# Comparison operators

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.<br>(a > b) is not 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 |

https://www.tutorialspoint.com/python3/python_basic_operators.htm

# Bitwise Operators

Bitwise operator performs bit-by-bit operation.
**Examples:**

a = 60 and b = 13 in binary format they will be
a = 0011 1100
b = 0000 1101

a&b = 0000 1100

a|b = 0011 1101

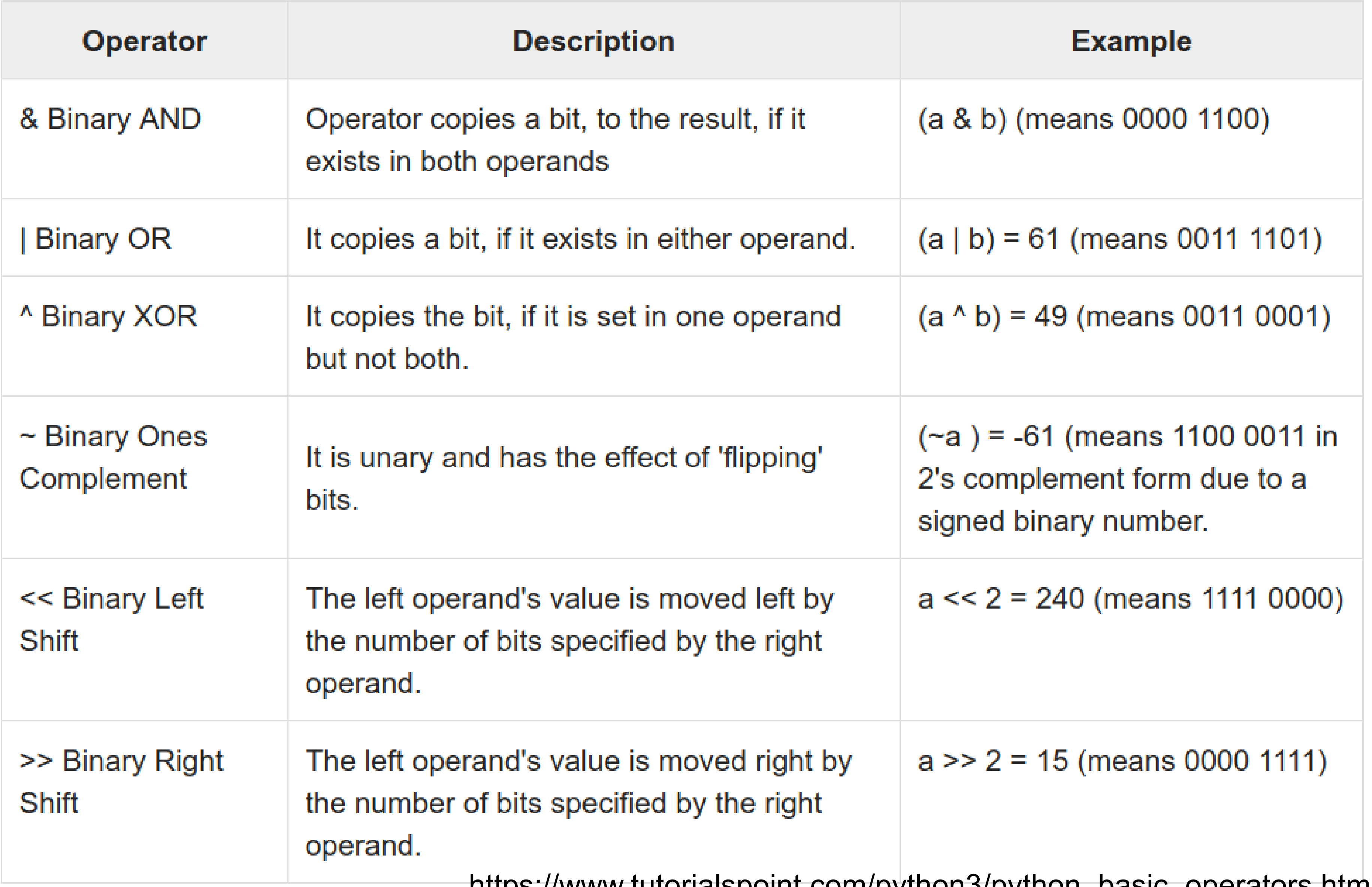
a^b = 0011 0001

~a = 1100 0011

Python's built-in function bin() can be used to obtain binary representation of an integer number.

https://www.tutorialspoint.com/python3/python_basic_operators.htm

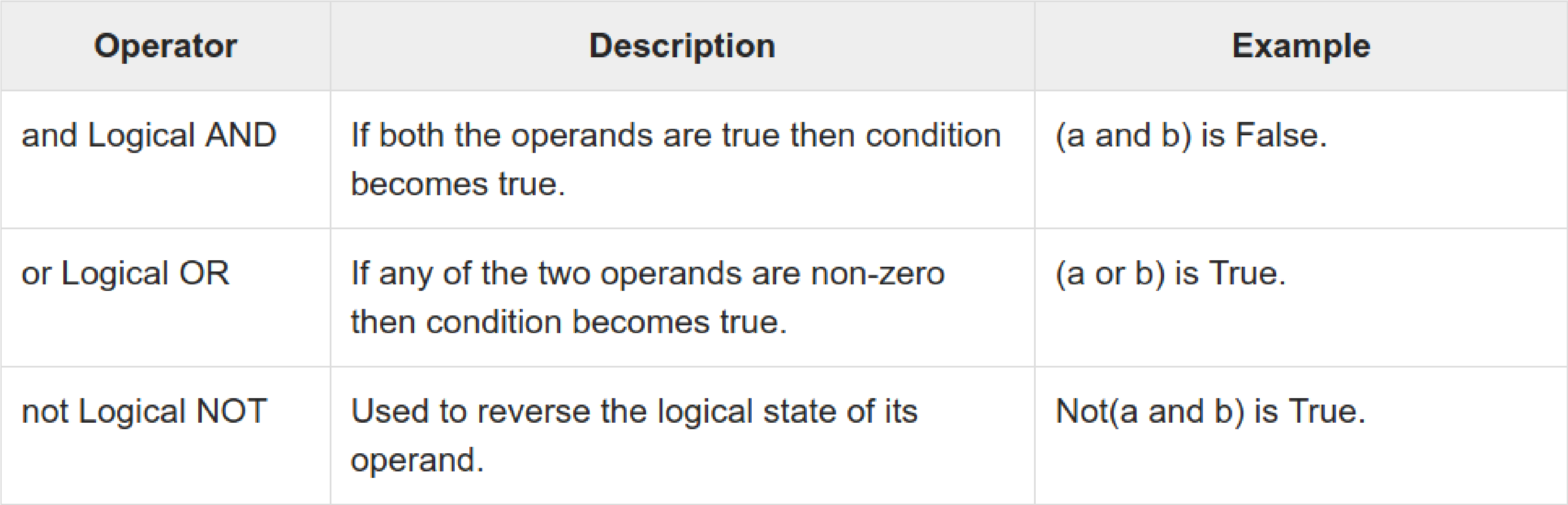
# Bitwise Operators



| Operator | Description | Example |
|---|---|---|
| & Binary AND | Operator copies a bit, to the result, if it exists in both operands | (a & b) (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 operand's 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 operand's value is moved right by the number of bits specified by the right operand. | a >> 2 = 15 (means 0000 1111) |

https://www.tutorialspoint.com/python3/python_basic_operators.htm

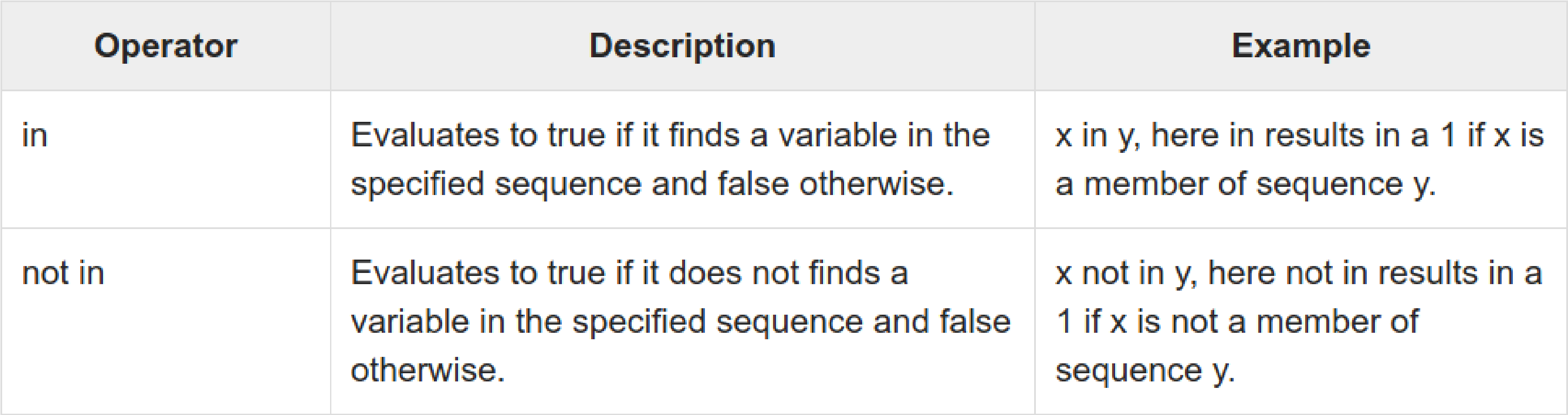
# Logical Operators



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

https://www.tutorialspoint.com/python3/python_basic_operators.htm

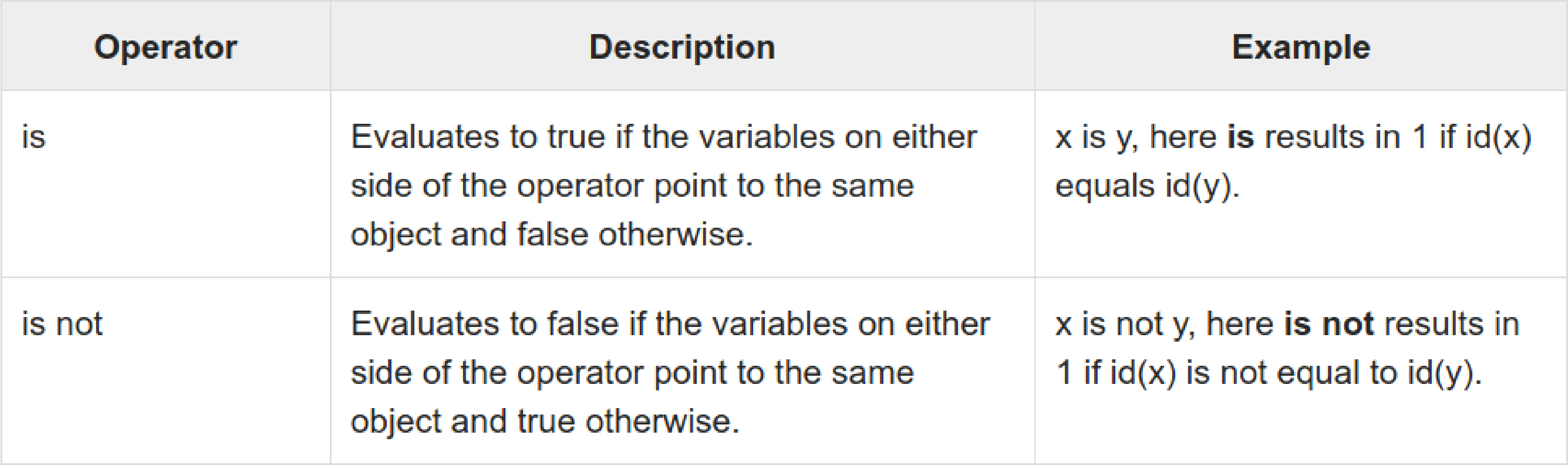
# Membership Operators



| 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. |

https://www.tutorialspoint.com/python3/python_basic_operators.htm

# Identity Operators



| 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). |

https://www.tutorialspoint.com/python3/python_basic_operators.htm

# Examples: Operations on numeric types

```
>>> k=5
>>> s=5+1j                # imaginary part cannot be only j
>>> type(s+k)             # imaginary number conversion
<type 'complex'>

>>> 4 and 2               # logical comparison
2

>>> 4 & 2                # bitwise comparison between the binaries 100 and 010
0

>>> 4 | 2               # bitwise comparison between the binaries 100 and 010
6
```

# Operations on numeric types: math module

The math module provides some of the more commons mathematical operations.
It does not work with complex numbers.
cmath module works for complex numbers.

The available functions are:

- Trigonometric functions: cos, sin, tan, asin, acos, atan, sinh, cosh, tanh.
- Exponentiaiton and logarithmic functions: pow, exp, log, log10, sqrt
- Angles representation and conversions: degrees, radians, ceil , floor, fabs

In the math module are defined the numerical constants pi and e

```
>>> import math
>>> math.pi
3.141592653589793
>>> math.e
2.718281828459045
```

# Bool type

Booleans True and False are available in Python.
bool is a subclass of int
True corresponds to 1
False corresponds to 0

Integer values can be used to reprsent boolean values with the following convention:
0 corresponds to False
all integer values greater than zero correspond to True

It is good practice to use the bool type to represent boolean values.

**Example**:

```
>>> a=1
>>> type(a)
<type 'int'>
>>> if(a):
print 'True'
True
>>> a=False
>>> type(a)
<type 'bool'>
```

# String type

Literal strings are character sequences enclosed in quotes, single or double. Creating strings is as simple as assigning a value to a variable.

Sequences of triple ‘double quotes’ or triple ‘single quotes’ can be used to assign strings spanning in more than one row or containing single or double quotes of the other type.

**Example 1:**

```
>>> a="""I am a string spanning in 3 rows,
... containing 'sigle quotes',
... containing "double quotes",
... containing '''triple quotes'''
... """
>>> print(a)
I am a string spanning in 3 rows,
containing 'sigle quotes',
containing "double quotes",
containing '''triple quotes'''
```

**Example 2:**

```
>>> b='''I am a string spanning in 3 rows,
... containing 'sigle quotes',
... containing "double quotes",
... containing """triple quotes"""
... '''
>>> print(b)
I am a string spanning in 3 rows,
containing 'sigle quotes',
containing "double quotes",
containing """triple quotes"""
```

# String type

- We can create strings by enclosing characters in quotes (single or double).

Creating strings is as simple as assigning a value to a variable.

**Example:**

```
var1 = 'Hello World!'
var2 = "Python Programming"
```

- To access substrings, use the square brackets for slicing along with the index or indices to obtain your substring.

**Example:**

```
#!/usr/bin/python3
var1 = 'Hello World!'
var2 = "Python Programming"

print ("var1[0]: ", var1[0])
print ("var2[1:5]: ", var2[1:5])
```

**Output:**

```
var1[0]:  H
var2[1:5]:  ytho
```

# String type

- The type char does not exists. A single char can be accesed using the operator [] or slicing the string with the operator [begin:end] (slicing)

**Example:**

```
>>> a = “Hello world”
>>> a[1]
‘e’
>>> a[1:2]
‘e’
```

- The single char can not be accessed, but a new value can be assigned to the string

**Example:**

```
>>> a=‘Primo valore’
>>> a = “Prima valore”      # Ok string re-assignment
>>> a[4] = ‘o’ #Errore      # NOT Ok single character assignment
  File "<stdin>", line 1
    a[4] = ‘o’
         ^
SyntaxError: invalid syntax
```

# String type: operators

- The operators + and * can be used for string operations.

Operators priority is maintained.

**Example**

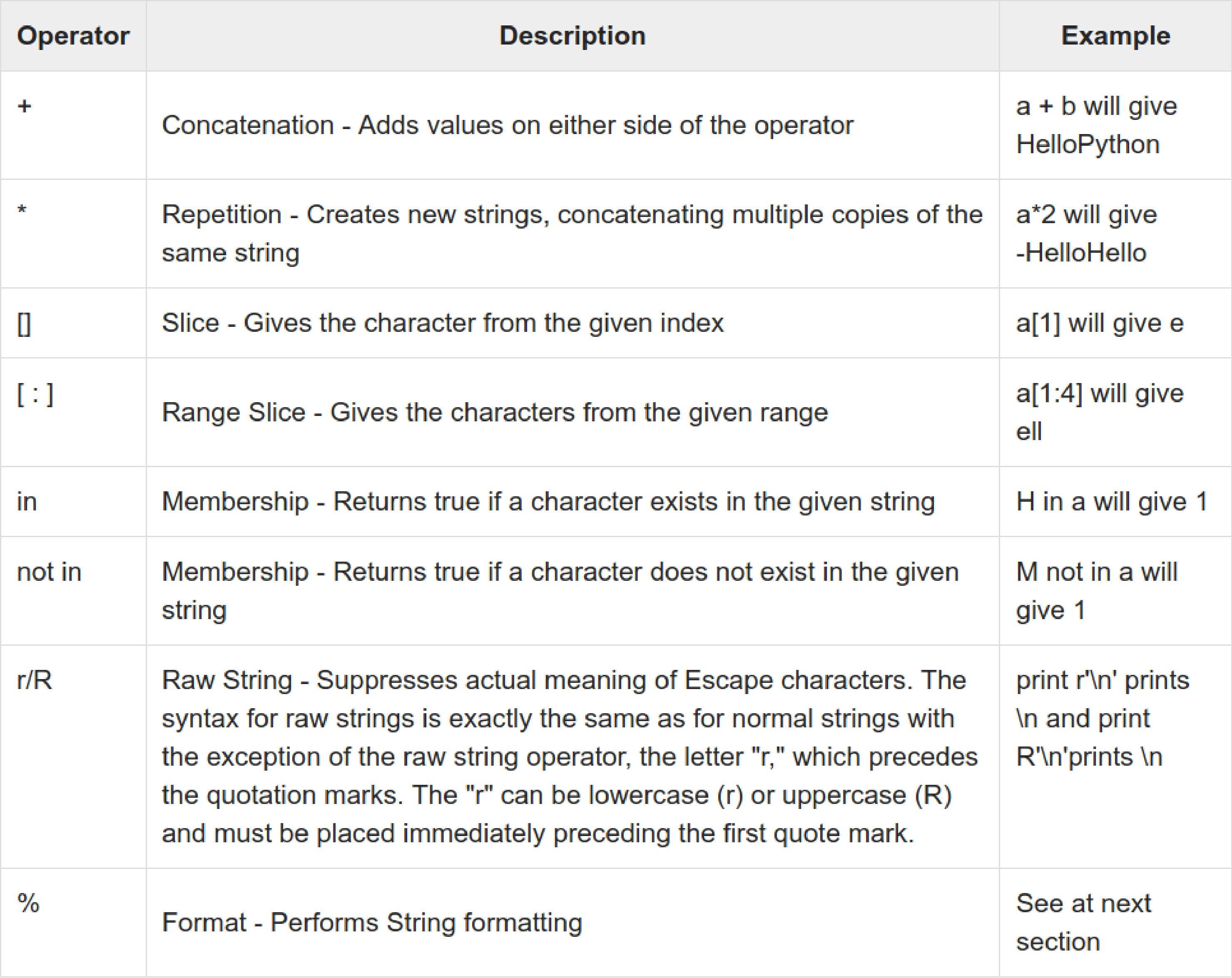
```
>>> a = ‘Hello’
>>> a+a+a                # Concatenation
‘HelloHelloHello’

>>> a = ‘He’+’l’*2+’o World’          # Multiple concatenation
>>> a
‘Hello World’
```

- It exists also the possibility to insert wherever in the string using the operator %

**Example**

```
name = "Peter"
my_string = "Hello %s" % name                        # Append or insert
my_string = "Hello %s, how are you? %s" % (name, 'ok')      # Insert multiple values
```

# String type: operators



| Operator | Description | Example |
|---|---|---|
| + | Concatenation - Adds values on either side of the operator | a + b will give HelloPython |
| * | Repetition - Creates new strings, concatenating multiple copies of the same string | a*2 will give -HelloHello |
| [] | Slice - Gives the character from the given index | a[1] will give e |
| [ : ] | Range Slice - Gives the characters from the given range | 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 |
| r/R | Raw String - Suppresses actual meaning of Escape characters. The syntax for raw strings is exactly the same as for normal strings with the exception of the raw string operator, the letter "r," which precedes the quotation marks. The "r" can be lowercase (r) or uppercase (R) and must be placed immediately preceding the first quote mark. | print r'\n' prints \n and print R'\n'prints \n |
| % | Format - Performs String formatting | See at next section |

# String type: escape characters

Escaping allows to add special characters inside a string.

**Example**

```
>>> a = ‘What’s your name?’ #Errore
SyntaxError: invalid syntax
>>> a = “What’s your name?” # Ok if I create the string with double quotes
>>> a = ‘What\’s your name?’ # Ok if you escape the single quote character
```

Most common escape characters in string manipulation:

| | | |
|---|---|---|
| • \t Tab | ‘Ciao\tciao!’ | Ciao ciao! |
| • \n New Line | ‘Ciao\nciao!’ | Ciao<br>ciao! |
| • \\ Backslash | ‘c:\\Programmi\\pp’ | c:\Programmi\pp |
| • \” Double quote | ‘Repeat: \”Hello\”’ | Repeat: ”Hello” |
| • \’ Single quote | “Repeat:\’Hello\’” | Repeat: ‘Hello’ |

# String type: row string

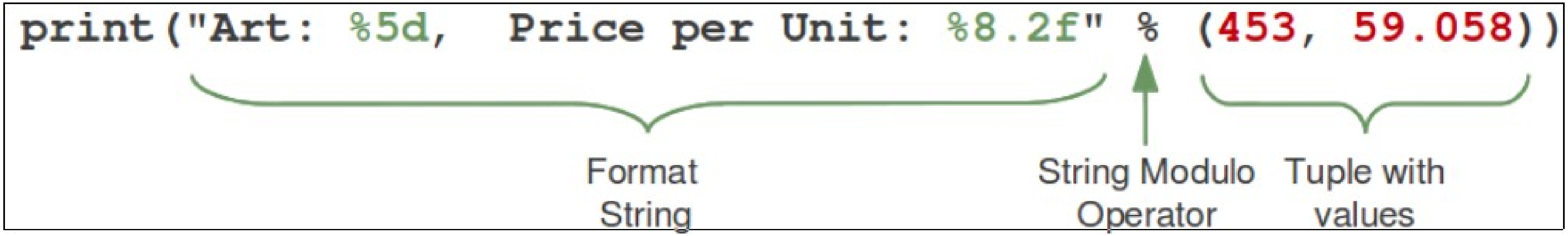
Row string is a string preceded by r or R in front of it.
In a row string a character preceded by \ is included without changes.

**Esempio**

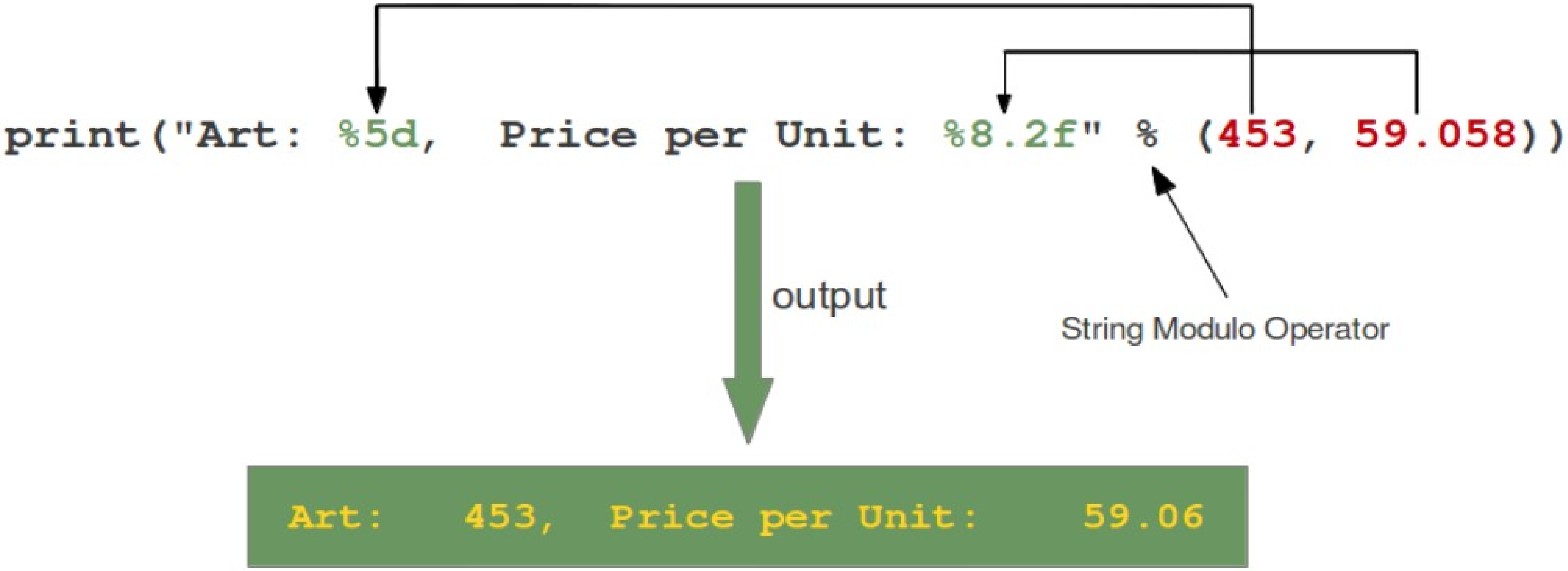
```
>>> a = r'Hello \t World'
#Raw string
>>> a
'Hello \t World'
```

# String type: format output

Python allows output formatting.
The % charactes has a special meaning when used in strings, because it is used to format output.



```
print("Art: %5d,  Price per Unit: %8.2f" % (453, 59.058))
```



```
print("Art: %5d,  Price per Unit: %8.2f" % (453, 59.058))
```

```
Art:   453,  Price per Unit:    59.06
```

# String type: format output

Ther are a lot of possibility to format output.

Examples:

%s, String<br>
%f, Floating point decimal format<br>
%c, Single character (accepts integer or single character string)<br>
%x, Unsigned hexadecimal (lowercase)<br>
%o, Unsigned octal<br>
%%, No argument is converted, results in a "%" character in the result<br>
%e, Floating point exponential format (lowercase)

Format output: https://www.python-course.eu/python3_formatted_output.php
https://www.tutorialspoint.com/python3/python_strings.htm

# String type: format output

**Example**

```
>>> ”Oggi è %s %d %s” % (“Venerdì’’,20,”Febbraio”)
>>> print _
Oggi è Venerdì 20 Febbraio
```

Already seen example:

```
name = "Peter"
my_string = "Hello %s" % name                          # Append and Insert
my_string = "Hello %s, how are you? %s" % (name, 'ok')    # Insert multiple values
```

# String type: built-in functions

Strings, as all the python objects, have a set of functionalities accessible with built-in Python functions (i.e. functions always available in the python interpreter).

- Manipulate: concat, split, characters deletion and unions.

-split([sep [,maxsplit]])
-replace (old, new[, count])
-strip([chars])

**Example**
**Split:**

```
>>> s='Ciao Mondo'
>>> s.split('o',1)
['Cia', ' Mondo‘]
```

**Replace:**

```
>>> s.replace('o','i',1)
'Ciai Mondo‘
```

**Strip:**

```
>>> s.strip('C')
'iao Mondo '
```

# String type: format built-in functions

Formattazione: align, upper case, lower case

-upper() e lower() e swapcase()

-center(width[, fillchar]) e ljust(width[, fillchar]) e rjust(width[,fillchar])

**Example**

```
>>> s = ‘Hello’

>>> s.center(10,’.’)
‘..Hello...’

>>> s.upper()
‘HELLO’
```

# String type: search built-in functions

- find(sub [,start [,end]])
- rindex(sub [,start [,end]]) returns the highest index of the substring inside the string (if found). If the substring is not found, it raises an exception.
- index(sub [,start [,end]])

- rfind(sub [,start [,end]]) returns the highest index of the substring (if found). If not found, it returns -1.

- count(sub[, start[, end]])
- isupper() returns whether or not all characters in a string are uppercased or not.
- islower() returns whether or not all characters in a string are lowercased or not.
- startswith(prefix[, start[, end]])
- endswith(prefix[, start[, end]])

# Container Data Types

- list []
- tuple ()
- dict {}
- set
- frozenset

# list[]

A list is initialized putting elements comma separated inside squared brackets.

- Items in a list can be of different type, both built-in and user defined.
- Indexes in a list start from zero.
- A list can be istantiated without specifing the list length or data type.

Single list elements can be accessed with the operator [ ]

**Example:**

```
>>>l=[]              # empty list instance
>>> print(l)
[]
>>>m=[‘Lista’,’di’,4,’elementi’]     # initialize a list
>>>print m[2],m[0]                    # access single list elements
>>>4 Lista
```

A list is an ordered sequence list, so the list items order is maintained.

# list[] : Basic List Operations

Lists respond to the operators
+ concatenation
* repetition
like strings, except that the result is a new list, not a string.

In fact, lists respond to all of the general sequence operations we saw on strings.

| Python Expression | Results | Description |
|---|---|---|
| len([1, 2, 3]) | 3 | Length |
| [1, 2, 3] + [4, 5, 6] | [1, 2, 3, 4, 5, 6] | Concatenation |
| ['Hi!'] * 4 | ['Hi!', 'Hi!', 'Hi!', 'Hi!'] | Repetition |
| 3 in [1, 2, 3] | True | Membership |
| for x in [1, 2, 3]: print x, | 1 2 3 | Iteration |

# list[] : slicing operator

List support the slicing operator [start:stop:step]

```
L = ['spam', 'Spam', 'SPAM!']
L[2]   → SPAM!              # Offsets start at zero
L[-2]  → Spam               # Negative: count from the right
L[1:]  →['Spam', 'SPAM!']   # Slicing fetches sections
```

**Example:**

```
>>>a=[0,1,2,3,4,5,6,7]
>>>a[0:6]
[0,1,2,3,4,5]
>>>a[1:6:2]
[1,3,5]
>>>a[1::2]          # no ‘stop’ means until the end of list
[1,3,5,7]
>>>a[::2]           # no ‘start’ means from the first item of the list
[0,2,4,6]
```

Slicing can be also negative

```
>>>a[6:0:-2]       # starts fom index 6, ends to index 0 going back with step 2
[6,4,2]
```

# list[] : range() built-in function

The range() function is used to generate lists of integer numbers.

**Syntax:**
range(start,stop,step) generates a list of integer from ‘start’ to ‘stop’ with interval ‘step’

**Example**

```
>>>a=range(3)
[0,1,2]

>>>type(a)
<type 'list'>

>>>a=range(1,10)
[0,1,2,3,4,5,6,7,8,9]

>>>a=range(1,10,2)
[1,3,5,7,9]
```

# list[] : Complex Operations

Lists supports complex operations.

**Examples:**

```
>>>a=range(10)
>>>b=[el*2 for el in a]
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

>>> l=[1,2]
>>> l2=['a','b']
>>> l3=[4,5]
>>> f=[(e1,e2,e3) for e1 in l for e2 in l2 for e3 in l3]
[(1, 'a', 4), (1, 'a', 5), (1, 'b', 4), (1, 'b', 5), (2, 'a', 4), (2, 'a', 5), (2, 'b', 4), (2, 'b', 5)]
```

This can be done also with:

```
>>> for e1 in l:
        for e2 in l2:
            for e3 in l3:
                f.append((e1,e2,e3))
```

# list[] : main functions

| **Function** | **Description** |
|---|---|
| cmp(list1, list2) | Compare elements of  lists |
| len(list) | Gives the total length of the list |
| max(list) | Returns item from the list with max value |
| min(list) | Returns item from the list with min value |
| list(seq) | Converts a tuple into list |

**Syntax** : cmp(list1, list2)

Parameters :
list1 : The first argument list to be compared.
list2 : The second argument list to be compared.

**Returns** : This function returns 1, if first list is “greater” than second list, -1 if first list is smaller than the second list else it returns 0 if both the lists are equal.

# list[] : main methods

List containers can be modified.
List objects contain built-in methods to modify members of a list.

| Method | Description |
|---|---|
| list.append(object) | Appends object to list |
| list.insert(index, object) | Inserts object obj into list at offset index |
| list.extend(seq) | Appends the contents of seq to list |
| list.pop(index) | Removes and returns last object or obj at index from list |
| list.remove(obj) | Removes object obj from list |
| list.count(value) | Returns count of how many times value occurs in list |
| list.index(obj) | Returns the lowest index in list where obj appears |
| list.reverse() | Reverses objects of list in place |
| list.sort([func]) | Sorts objects of list, use compare func if given |

# list[] : main methods exercises

Practice with the list methods proposed in the previous slide

# list[] : about efficiency

The operators concatenation + (or +=) and repetition * are supported by lists.
The operator + and the function extend() have the same functionality, but different execution time (efficiency)

**Example:**

```
import time
l=range(100000000)
v=range(1000000)
T1=time.clock()
s=l+v
T2=time.clock()
print(‘ + execution time: :’ , T2-T1, ‘s’)
, “s”
T3=time.clock()
l.extend(v)
T4=time.clock()
print(‘extend execution time:’, T4-T3 , ‘s’)
```

**Output:**

```
 + execution time: 2.81 s
 extend execution time: 0.033 s
```

# list[] for queue and stack

List can be easily used as stack or queue.
pop and append methods can be used to implement the LIFO logic typical  of stacks.
pop with index 0 and append can be used to implement the FIFO logic typical of queue.

**Example:**

```
stack=[1, 2, 3, 4]
print(‘Initial Stack : ’, stack)
for i in range(5,7):
    stack.append(i)
print (“Append: “, stack)
stack.pop()
print (“Pop: ” , stack)

queue=[ ‘a’,’b’,’c’,’d’ ]
print(“Initial Queue : ”, queue)
queue.append(‘e’)
queue.append(‘f’)
print(“Append : ”, queue)
queue.pop(0)
print(“Pop : ”, queue)
```

**Output:**

```
Initial Stack : [1, 2, 3, 4]
Append: [1, 2, 3, 4, 5, 6]
Pop: [1, 2, 3, 4, 5]
Initial Queue :  ['a', 'b', 'c', 'd']
Append: ['a', 'b', 'c', 'd', 'e', 'f']
Pop: ['b', 'c', 'd', 'e', 'f']
```

# tuple()

A tuple is a sequence ordered data enclosed between ().
Tuples are sequences, just like lists. The differences between tuples and lists are,

- the tuples cannot be changed unlike lists, tuple are immutable.
- tuples use parentheses, whereas lists use square brackets.

A tuple is created putting in it different comma-separated values. Optionally, can be put these comma-separated values between parentheses also.

**Example:**

```
tup1 = "a", "b", "c", "d";
tup2 = ('physics', 'chemistry', 1997, 2000);    # Data in a tuple can be heterogeneous
tup3 = (1, 2, 3, 4, 5 );
```

The empty tuple is written as two parentheses containing nothing. **Example:**

```
tup1 = ();
```

A tuple containing a single value must be written including a comma. **Example:**

```
tup1 = (50,);
```

Tuple indices start at 0, like string indices.

# tuple() : Accessing Values in Tuples

To access values in tuple, use the square brackets for access the single element. slicing [start:end] is also available to obtain value available at that index.

**Example**

```
tup1 = ('physics', 'chemistry', 1997, 2000)
tup2 = (1, 2, 3, 4, 5, 6, 7)
print("tup1[0]: ", tup1[0])          # access to single element
print("tup2[1:5]: ", tup2[1:5])      # access to slice
```

**Output:**

```
tup1[0]:  physics
tup2[1:5]:  [2, 3, 4, 5]
```

- tuple are immutable, so does NOT contain methods to:
  - eliminate elements
  - insert elements
- 'tuple' object does not support item assignment

  **Example:**

  ```
  >>>t1=(1,2,3,4,’ciao’,’mondo’,[2,3])
  >>>t1[3]=‘jkjk’
  ```

**Output:**

```
Traceback (most recent call last):
      File "<pyshell#26>", line 1, in
<module>
      t1[1]=3
      TypeError: 'tuple' object does not
support item assignment
```

# tuple() : Delete

Removing individual tuple elements is not possible.
It can be created a new tuple with the undesired elements discarded.

To explicitly remove an entire tuple, just use the del statement.

**Example**

```
tup = ('physics', 'chemistry', 1997, 2000)
print tup
del tup
print("After deleting tup : ")
print(tup)
```

This produces an exception raised, because after del tup tuple does not exist any more

**Output:**

```
('physics', 'chemistry', 1997, 2000)
After deleting tup :
Traceback (most recent call last):
   File "test.py", line 9, in <module>
      print tup;
NameError: name 'tup' is not defined
```

# tuple() : basic operations

Tuples respond to the + and * operators much like strings;
+ means concatenation
* means repetition
the result is a new tuple, not a string.

Tuples respond to all of the general sequence operations availble on strings:

| Python Expression | Results | Description |
|---|---|---|
| len((1, 2, 3)) | 3 | Length |
| (1, 2, 3) + (4, 5, 6) | (1, 2, 3, 4, 5, 6) | Concatenation |
| ('Hi!',) * 4 | ('Hi!', 'Hi!', 'Hi!', 'Hi!') | Repetition |
| 3 in (1, 2, 3) | True | Membership |
| for x in (1, 2, 3): print x, | 1 2 3 | Iteration |

# tuple() : Built-in Tuple Functions

| Function | Description |
|---|---|
| cmp(tuple1, tuple2) | Compares elements of both tuples |
| len(tuple) | Gives the total length of the tuple |
| max(tuple) | Returns item from the tuple with max value |
| min(tuple) | Returns item from the tuple with min value |
| tuple(seq) | Converts a list into tuple |

# dict{}

Python dictionary is an unordered collection of items.

Elements in a dictionary are key:value pairs.

- values can be of any data type and can repeat,
- keys must be of immutable type (string, number or tuple with immutable elements) and must be unique. Keys are case-sensitive

Each element in a dictionary is identified by the key.
Dictionaries are optimized to retrieve values when the key is known.

**Example** how to create a dictionary:

```
>>>d={ }                              # empty dictionary
>>>d={1: ’Hello’, ‘due’: ’World’}     # dictionary with two elements
>>>d[1]                               # access to a dictionary element
‘hello’
```

# dict{} : Creation examples

```
# empty dictionary
my_dict = {}

# dictionary with integer keys
my_dict = {1: 'apple', 2: 'ball'}

# dictionary with mixed keys
my_dict = {'name': 'John', 1: [2, 4, 3]}

# using dict()
my_dict = dict({1:'apple', 2:'ball'})

# from sequence having each item as a pair
my_dict = dict([(1,'apple'), (2,'ball')])
```

# dict{} : Access elements

In the other container types indexing is used to access values,
Dictionary uses keys to access values.
Key can be used either inside square brackets or with the get() method.

get()  returns None if the key is not found.
[ ] returns KeyError if the key is not found.

**Example:**

```
my_dict = {'name':'Jack', 'age': 26}

print(my_dict['name'])        # Output: Jack

print(my_dict.get('age'))     # Output: 26

# Trying to access keys which doesn't exist throws error (try)
# my_dict.get('address')
# my_dict['address']
```

- keys() and values() functions return respectively the keys and the values present in a dictionary.

# dict{} : change or add elements in a dictionary

Dictionary are mutable. We can add new items or change the value of existing items using assignment operator.

If the key is already present, value gets updated,
else a new key: value pair is added to the dictionary.

**Example:**

```
my_dict = {'name':'Jack', 'age': 26}

# update value
my_dict['age'] = 27

#Output: {'age': 27, 'name': 'Jack'}
print(my_dict)

# add item
my_dict['address'] = 'Downtown'

# Output: {'address': 'Downtown', 'age': 27, 'name': 'Jack'}
print(my_dict)
```

# dict{}: delete or remove elements

We can remove a particular item in a dictionary by using the method pop(). This method removes as item with the provided key and returns the value.

The method, popitem() can be used to remove and return an arbitrary item (key, value) form the dictionary.

All the items can be removed at once using the clear() method.

del keyword can be used to remove individual items or the entire dictionary itself.

# dict{}: delete or remove elements examples

```
# create a dictionary
squares = {1:1, 2:4, 3:9, 4:16, 5:25}

# remove a particular item
print(squares.pop(4))              # Output: 16
print(squares)                     # Output: {1: 1, 2: 4, 3: 9, 5: 25}

# remove an arbitrary item
print(squares.popitem())           # Output: (1, 1)
print(squares)                     # Output: {2: 4, 3: 9, 5: 25}

# delete a particular item
del squares[5]
print(squares)                     # Output: {2: 4, 3: 9}

# remove all items
squares.clear()
print(squares)                     # Output: {}

# delete the dictionary itself
del squares

# print(squares)                   # Throws Error
```

# dict{}: built-in functions

| Function | Description |
|---|---|
| all() | Return True if all keys of the dictionary are true (or if the dictionary is empty). |
| any() | Return True if any key of the dictionary is true. If the dictionary is empty, return False. |
| len() | Return the length (the number of items) in the dictionary. |
| cmp() | Compares items of two dictionaries. |
| sorted() | Return a new sorted list of keys in the dictionary. |

**Example:**

```
squares = {1: 1, 3: 9, 5: 25, 7: 49, 9: 81}

print(len(squares))          # Output: 5
print(sorted(squares))       # Output: [1, 3, 5, 7, 9]
```

**Exercise:** Practice with these functions

# dict{}: built-in methods

| Method | Description |
|---|---|
| clear() | Remove all items form the dictionary. |
| copy() | Return a shallow copy of the dictionary. |
| fromkeys(seq[, v]) | Return a new dictionary with keys from seq and value equal to v (defaults to None). |
| get(key[,d]) | Compares items of two dictionaries. |
| items() | Return a new sorted list of keys in the dictionary. |
| keys() | Return a new view of the dictionary's keys. |
| pop(key[,d]) | Remove the item with key and return its value or d if key is not found. If d is not provided and key is not found, raises KeyError. |
| popitem() | Remove and return an arbitary item (key, value). Raises KeyError if the dictionary is empty. |
| setdefault(key[,d]) | If key is in the dictionary, return its value. If not, insert key with a value of d and return d (defaults to None). |
| update([other]) | Update the dictionary with the key/value pairs from other, overwriting existing keys. |
| values() | Return a new view of the dictionary's values |
| has_key(k) | Return True or Falese if key is in the dictionary |

# dict{}: Built-in methods example

```
marks = {}.fromkeys(['Math','English','Science'], 0)


print(marks)              # Output: {'English': 0, 'Math': 0, 'Science': 0}


for item in marks.items():
    print(item)
list(sorted(marks.keys()))     # Output: ['English', 'Math', 'Science']
```

# dict{}: Other operations

### Iterating Through a Dictionary

Using a for loop we can iterate though each key in a dictionary.

**Example:**

```
squares = {1: 1, 3: 9, 5: 25, 7: 49, 9: 81}
for i in squares:
    print(squares[i])
```

### Dictionary Membership Test

We can test if a key is in a dictionary or not using the keyword in. Notice that membership test is for keys only, not for values.

**Example:**

```
squares = {1: 1, 3: 9, 5: 25, 7: 49, 9: 81}

print(1 in squares)            # Output: True
print(2 not in squares)        # Output: True

# membership tests for key only not value
print(49 in squares)           # Output: False
```

# dict{}: Python dictionary comprehension

Dictionary comprehension is an elegant and concise way to create new dictionary from an iterable in Python.

Dictionary comprehension consists of an expression pair (key: value) followed by for statement inside curly braces {}.

**Example** to make a dictionary with each item being a pair of a number and its square.

```
squares = {x: x*x for x in range(6)}
print(squares)          # Output: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

Equivalent to:

```
odd_squares = {x: x*x for x in range(11) if x%2 == 1}
print(odd_squares)  # Output: {1: 1, 3: 9, 5: 25, 7: 49, 9: 81}
```

A dictionary comprehension can optionally contain more for or if statements. An optional if statement can filter out items to form the new dictionary.

**Example** to make dictionary with only odd items.

```
odd_squares = {x: x*x for x in range(11) if x%2 == 1}
print(odd_squares)        # Output: {1: 1, 3: 9, 5: 25, 7: 49, 9: 81}
```

# set - fronzenset

Python has two structures to represent sets of elements:

- set is a mutable, unordered collection of etherogeneous objects
- frozenset is an immutable, unordered collection of etherogeneous objects. It is a freezed set

In both cases, elements are unique.

**Example:**

```
>>>s=set((‘ciao’,1,’Mondo’))
>>>fs=frozenset((‘ciao’,2))
```

- Sets provide methods to modify the data set:
  - insert with add(obj)
  - modify with update(obj)

**Example:**

```
>>>s=set(('abc','def',1,2,3,'ghi'))
>>>s.add(4)
>>>s.update((‘lmn’,5))
>>>s
set([1, 2, 3, 4, 5, 'abc', 'lmn', 'ghi', 'def'])
```

# set - frozenset

- Removal
  - discard(x)
  - remove(x)
  - clear()
  - pop()

**Example**:

```
>>> s=set([2, 3, 'abc', 'ghi', 'def'])
>>> s
set([2, 3, 'abc', 'def', 'ghi'])
>>> s.remove(3)
>>> s.discard(2)
>>> s.pop()
'abc'
>>> s.clear(); s
set([])
```

# set - frozenset

- Both containers contain methods to manage operations :
  - union,
  - intersection,
  - difference,
  - issubset,
  - issuperset

**Example:**

```
>>>s=set((1,2))
>>>s2=frozenset((2,3,4))
>>>s3=s.union(s2)
>>>s4=s.difference(s2)
>>>s5=s2.intersection(s)
>>>s.issubset(s2)
False
>>>print(’s3’, s3 , ’s4’, s4, ’s5’, s5)
s3
s3 {1, 2, 3, 4} s4 {1} s5 frozenset({2})
```

In both cases data can be of different types.

=> frozenset are immutable, so they can be used to index dictionaries

# The Anonymous Functions or Lambdas

Anonymous functions or lambdas are small functions which do not need a name (i.e., an identifier).
In Python an anonymous function has 3 parts:

- The lambda keyword, used in place of the keyword ‘def’ used for generic functions
- A set of parameters (can take any number of parameters)
- The function body, which can contain only one expression (in one line of code).

**Syntax:**

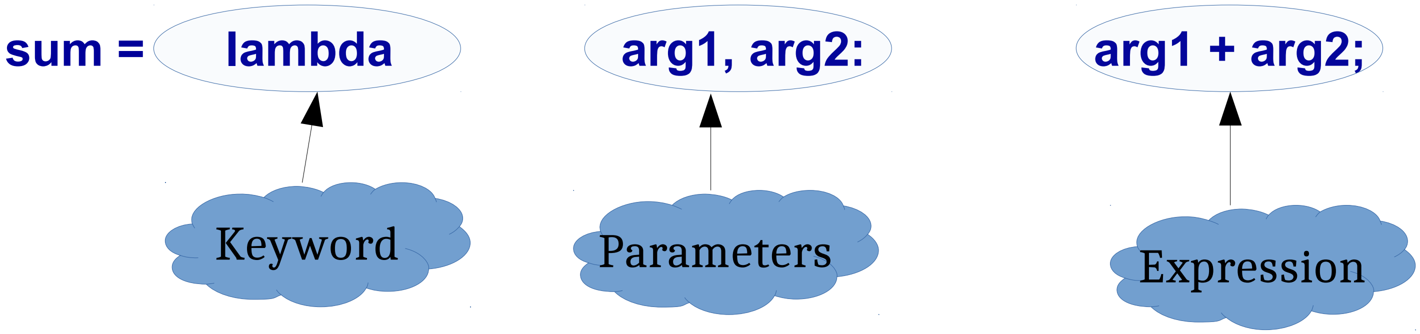
```
lambda [arg1 [,arg2,.....argn]]:expression
```

Features:

- The lambda function return just one value in the form of an expression.
- The lambda function cannot be a direct call to print because lambda requires an expression
- Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.

# Example 1: The Lambda/Anonymous Functions

```
#!/usr/bin/python

# Function definition is here
```



```
sum = lambda arg1, arg2: arg1 + arg2;
```

```
# Now you can call sum as a function
print "Value of total : ", sum( 10, 20 )
print "Value of total : ", sum( 20, 20 )
```

When the above code is executed, it produces the following result:

```
Value of total :  30
Value of total :  40
```

# Example 2: The print and lambda function (1)

**Code:**

```
string=’Hello World!’
print(lambda string : print(string))
```

**Output**:

```
$ python3
Python 3.6.9 (default, Apr 18 2020, 01:56:04)
[GCC 8.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> string='Hello World!'
>>> print(lambda string : print(string))
<function <lambda> at 0x7fe0922ebd90>
```

Works with python3 where print is a function (and a function application is an expression, so it will work in a lambda). In python2 print is a statement and this example does not work.

**Explanation**:

Define a string

Declare a lambda that calls a print statement

prints the result, passing the string as parameter.

Why doesn't the program print the string we pass?

Because the lambda itself returns a function object.

The external print instruction prints the result of the lambda function, i.e. the function object and the memory location where it is stored.

# Input parameters

A script can require one or more input parameters.

There are different ways to provide input parameters to a script:

- by command line
- by user
- by an input file

# Input parameters by command line.sys.argv

A script requiring parameters can be executed with:

```
$ python script.py param_1 param_2 param_3 …… param_n
```

- The argv[*] provided by tye sys module can be used to read the input parameters:
  - – argv[0]: contains the script name
  - – argv[1]:  param_1
  - – …...
  - – argv[i]:  param_i

# Example: command line input (try)

```
# script reqiring 2 input parameters
import sys

usage="""Requires two parameters (param1, param2)
Usage: python script.py param1 param2"""

if len(sys.argv) < 3:
    print('The script: ',sys.argv[0],usage)
    sys.exit(0) # exits after help printing

# read the two input parameters
param1 = sys.argv[1]
param2 = sys.argv[2]

# output the read parameters
print('''The two parameters  received as input
for the script are:\n ''',param1, param2)
```

# Input parameters user provided

The input parameters provided by the user can be read from the standard input (stdin) using the function input()

Example (try):

```
# the script takes from the user two input parameters
import sys

while(True):
    print('PLEASE INSERT AN INTEGER NUMBER IN THE RANGE 0-10')
    param1 = input()
    if int(param1) in range(11):
        while(True):
            print( 'PLEASE INSERT A CHAR PARAMETER IN [A,B,C]')
            param2 = input()
            if param2 in ['A','B','C']:
                print('uso I due parametri passati dall utente: ',param1,param2)
                sys.exit()
            else: print('TRY AGAIN PLEASE')
    else: print('TRY AGAIN PLEASE')
```

# Input parameters from file

```
infile='mydata.dat'
outfile='myout.dat'

indata = open( infile, 'r')
linee=indata.readlines()
indata.close()
processati=[ ]
x=[ ]
for el in linee:
    valori = el.split()
    x.append(float(valori[0])); y = float(valori[1])
    processati.append(f(y))

outdata = open(outfile, 'w')
i=0
for el in processati:
    outdata.write('%g %12.5e\n' % (x[i],el))
    i+=1
outdata.close()
```

```
def f(y):
    if y >= 0.0:
        return y**5*math.exp(-y)
    else:
        return 0.0
```

```
cat mydata.dat
2      16
13      5
19.3 11
```

Format output: https://www.python-course.eu/python3_formatted_output.php

# Input parameters from file

You can read the file with file.read()

```
file = open('.env', "r")
filecontent = file.read()
print("File content:")
print(filecontent)
my_line = ""

for line in filecontent.splitlines():
    print("Working on line", line)
    if line.find("DB_DATABASE="):
        print("Found line containing DB_DATABASE=")
        break
```

Source file:

cat .env

```
DB_HOST= http://localhost/
DB_DATABASE= bheng-local
DB_USERNAME= root
DB_PASSWORD= 1234567890
UNIX_SOCKET= /tmp/mysql.sock
```

Next lesson will go deeply on structured data and how to red them from files

# The if statement

The **if** statement is used for conditional execution: if a condition is true, we run a block of statements (called the if-block), else we process another block of statements (called the else-block).

The else clause is optional.

**Syntax:**

```
if test_expression :
    statement(s)
```

or

```
if test_expression :
    body of if
else:
    body of else
```

```
if test_expression1 :
    body of if
elif test_expression2 :
    body of elif
else:
    body of else
```

switch-case
simulation

# Control flow instructions

- Choose
- Cicle

# The if statement: example

```
number = 23
guess = int(input('Enter an integer : '))

if guess == number:
    # New block starts here
    print('Congratulations, you guessed it.')
    print('(but you do not win any prizes!)')
    # New block ends here
elif guess < number:
    # Another block
    print('No, it is a little higher than that')
    # You can do whatever you want in a block ...
else:
    print('No, it is a little lower than that')
    # you must have guessed > number to reach here

print('Done')
# This last statement is always executed,
# after the if statement is executed.
```

# The while statement

The **while** statement allows you to repeatedly execute a block of statements as long as a condition is true.

A while statement is an example of what is called a looping statement.

A while statement can have an optional else clause. If the else clause is present, it is always executed once after the while loop is over unless a break statement is encountered.

**Syntax:**

```
while test_condition :
    while-statement(s)
[else:
    else-statement(s)]
```

else clause is optional

# The while statement: example

```
number = 23
running = True

while running:
    guess = int(input('Enter an integer : '))

    if guess == number:
        print('Congratulations, you guessed it.')
        # this causes the while loop to stop
        running = False
    elif guess < number:
        print('No, it is a little higher than that.')
    else:
        print('No, it is a little lower than that.')
else:
    print('The while loop is over.')
    # Do anything else you want to do here
print('Done')
```

# The for statement

The **for** statement is a looping statement which iterates over a sequence of objects, i.e. go through each item in a sequence. A sequence is just an ordered collection of items.

In general we can use any kind of sequence of any kind of objects.

An else clause is optional, when included, it is always executed once after the for loop is over unless a break statement is encountered.

**Syntax:**

```
for iterating_var in sequence:
    statements(s)
[else:
    else-statement(s)]
```

else clause is optional

**Example:**

```
for i in range(1, 5):
    print(i)
else:
    print('The for loop is over')
```

# The break statement

The **break** statement is used to break out of a loop statement i.e. stop the execution of a looping statement, even if the loop condition has not become False or the sequence of items has not been completely iterated over.

An important note is that if you break out of a for or while loop, any corresponding loop else block is not executed.

**Example** (break.py)**:**

```
while True:
    s = input('Enter something : ')
    if s == 'quit':
        break
    print('Length of the string is', len(s))
print('Done')
```

# Exercise

Try a for and a while loop with an else clause verifying that the else clause is always executed except in case a break statement is found.

# The continue statement

The **continue** statement is used to tell Python to skip the rest of the statements in the current loop block and to continue to the next iteration of the loop.

**Example:**

```
while True:
    s = input('Enter something : ')
    if s == 'quit':
        break
    if len(s) < 3:
        print('Too small')
        continue
    print('Input is of sufficient length')
    # Do other kinds of processing here…
```

=> the continue statement works with the for loop as well.

# The pass statement

The **pass** statement does nothing. It can be used when a statement is required syntactically but the program requires no action.

**Example**:

```
>>> while True:
...     pass  # Busy-wait for keyboard interrupt (Ctrl+C)
```

- This is commonly used for creating minimal classes:

```
>>> class MyEmptyClass:
...     pass
```

- Another place pass can be used is as a place-holder for a function or conditional body when you are working on new code, allowing you to keep thinking at a more abstract level. The pass is silently ignored:

```
>>> def initlog():
            pass   # Remember to implement this!
```

# Exercise: control_flow_step0

Prepare a python script where all the presented examples on flow control statements are converted in functions.

Write a main block of code printing instructions and explanations useful to the user and then calling the functions.

Example of expected output:

This is if statement usage example.

You have to guess the right number trying repetitively:
Enter an integer :

…….
This is while statement usage example.

…….

*and so on…...*

# Exercise: control_flow_step1

Complicate the previous script giving the user the ability to choose which statement he likes to try.

Output example:

```
Choose if try
1. if statement
2. while statement
3. for statement
make your choose entering the number (1 or 2 or 3)
……..
```

# Exercise: control_flow_step2

Complicate the previous script giving the user the ability to choose how much iteration execute in case it is trying the for statement

Output example:

```
Choose if try
1. if statement
2. while statement
3. for statement
make your choose entering the number (1 or 2 or 3)
3
Enter how much iteration you want execute (integer)
```

# Exercise: control_flow_step3

Complicate the previous script giving the user the ability to choose repeatedly the control statement to test.

# Exercise: control_flow_step4

Complicate one of the previous scripts giving the user the ability to choose the reference number used for comparison in if and while statements (fixed to guess=23 in the already done exercices).

# Iterators

**for** cicle is generally used to iterate on iterable types like list, tuple, string, and in general containers.

Iterable types contain an object called iterator used by the for operator to iterate in the container.

The iterator object contains a next() method, returning the first available data in the container, useful to iterate in the container.

# Iterators. examples

```
>>> a = iter(list(range(10)))
>>> for i in a:
...    print(i)
0
1
2
3
4
5
6
7
8
9
```

```
>>> a = iter(list(range(10)))
>>> for i in a:
...    next(a)
...
1
3
5
7
9
```

```
>>> for i in a:
...      print("Printing: %s" % i)
...      next(a)
...
Printing: 0
1
Printing: 2
3
Printing: 4
5
Printing: 6
7
Printing: 8
9
>>>
```

# Data sequences and cicles

**for** cicle allows to iterate on every kind of iterable object like list, tuple, string, set, dictionary.

**Example:**

**LIST**

```
>>> a=[1,2,3,4,5]
>>> for el in a:
        print(el)
1
2
3
4
5
```

**STRING**

```
>>> a=‘’’Ciao’’’
>>> for el in a:
        print(el)
C
i
a
o
```

**SET**

```
>>>a=set([1,2,3,4])
>>> for el in a:
        print(el)
1
2
3
4
```

# Data sequences and cicles

**for** cicle allows to iterate on every kind of iterable object like list, tuple, string, set, dictionary.

**Example:**

**DICTIONARY (by key)**

```
>>> a={1:’a’,2:’b’}
>>> for el in a.keys():
        print(el)
1
2
```

**DICTIONARY(by value)**

```
>>> a={1:’a’,2:’b’}
>>> for el in a.values():
        print(el)
a
b
```

**DICTIONARY(by key-val)**

```
>>> a={1:’a’,2:’b’}
>>> for k,v in a.items():
        print(k,v)
1 a
2 b
```

**DICTIONARY**

```
>>> a={1:’a’,2:’b’}
>>> for el in a:
        print(el)
1
2
```

**DICTIONARY**

```
>>> a={1:’a’,2:’b’}
>>> for el in (1,2,3):
        print(a.get(el))
a
b
None
```

# range() function

The function range take in three arguments in total, however two of them are optional. The arguments are "start", "stop" and "step". "start" is what integer you'd like to start your list with, "stop" is what integer you'd like your list to stop at, and "step" is what your list elements will increment by.

```
>>> for i in range(1, 10, 2):
...     print(i)
...
1
3
5
7
9
```

**Syntax**:

```
range(stop)
range(start, stop[, step])
```

Python's range with Negative Numbers:

```
>>> for i in range(-1, -10, -1):
...     print(i)
...
-1
-2
-3
-4
-5
-6
-7
-8
-9
```

*you must do it this way for negative lists. Trying to use range(-10) will not work because range uses a default "step" of one.*

Note that if "start" is larger than "stop", the list returned will be empty. Also, if "step" is larger that "stop" minus "start", then "stop" will be raised to the value of "step" and the list will contain "start" as its only element.

**Example**:

```
>>> for i in xrange(70, 60):
...         print(i)
...
# Nothing is printed
>>> for i in xrange(10, 60, 70):
...         print(i)
...
10
```

# Read text file in matrix

Input1.txt:

```
0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0
0,0,2,1,0,2,0,0,0,0
0,0,2,1,1,2,2,0,0,1
0,0,1,2,2,1,1,0,0,2
1,0,1,1,1,2,1,0,2,1
```

Code to read file:

```
l = []
with open('input.txt', 'r') as f:
  for line in f:
    line = line.strip()
    if len(line) > 0:
      l.append(map(int, line.split(',')))
print(l)
```

# Read text file in matrix

Input1.txt:

```
0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0
0,0,2,1,0,2,0,0,0,0
0,0,2,1,1,2,2,0,0,1
0,0,1,2,2,1,1,0,0,2
1,0,1,1,1,2,1,0,2,1
```

Code to read file:

```
fin = open('input.txt','r')
a=[]
for line in fin.readlines():
    a.append( [ int (x) for x in line.split(',') ] )
```

```
l = []
with open('input.txt', 'r') as f:
  for line in f:
    line = line.strip()
    if len(line) > 0:
      l.append(map(int, line.split(',')))
print(l)
```

```
fin = open('input.txt','r')
a=[]
for line in fin.readlines():
    a.append( [ int (x) for x in line.split(',') ] )
```

# Read text file in matrix using numpy

Input1.txt:

```
0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0
0,0,2,1,0,2,0,0,0,0
0,0,2,1,1,2,2,0,0,1
0,0,1,2,2,1,1,0,0,2
1,0,1,1,1,2,1,0,2,1
```

Code to read file:

```
import numpy as np
input = np.loadtxt("input.txt", dtype='i',
delimiter=',')
print(input)
```

numpy is a library

numpy.loadtxt(fname, dtype=<class 'float'>, comments='#', delimiter=None, converters=None, skiprows=0, usecols=None, unpack=False, ndmin=0, encoding='bytes', max_rows=None)
[source]
Load data from a text file.

Each row in the text file must have the same number of values.

https://docs.scipy.org/doc/numpy/reference/generated/numpy.loadtxt.html

# Read text file in matrix: example

Input2.txt:

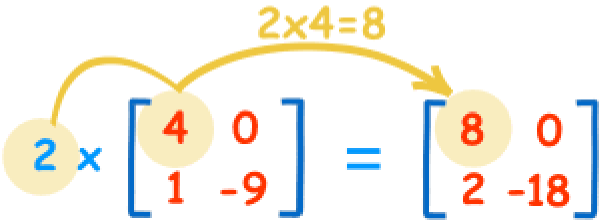
```
"0","0","0","0","1","0"
"0","0","0","2","1","0"
```

Code to read file:

```
with open('Input2.txt', 'r') as f:
    data = f.readlines() # read raw lines into an array

cleaned_matrix = []
for raw_line in data:
    split_line = raw_line.strip().split(",") # ["1", "0" ... ]
    nums_ls = [int(x.replace('"', '')) for x in split_line] # get rid of the quotation marks and convert to int
    cleaned_matrix.append(nums_ls)

print(cleaned_matrix)
```

# Multiply matrices: Matrix Multiply Constant

To multiply a matrix by a single number is easy:

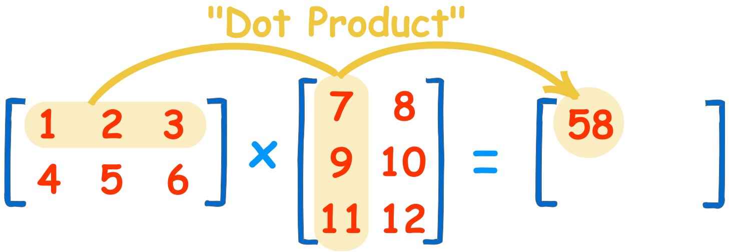


These are the calculations:
2×4=8 2×0=0
2×1=2 2×-9=-18

We call the number ("2" in this case) a scalar, so this is called "scalar multiplication".

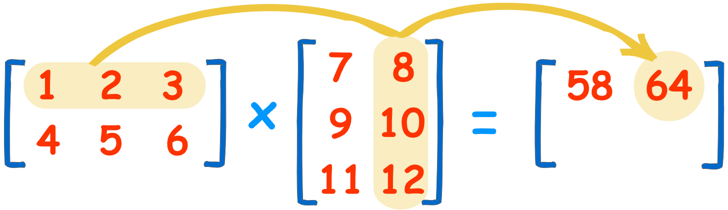
https://www.mathsisfun.com/algebra/matrix-multiplying.html

# Multiply matrices: Multiplying a Matrix by Another Matrix



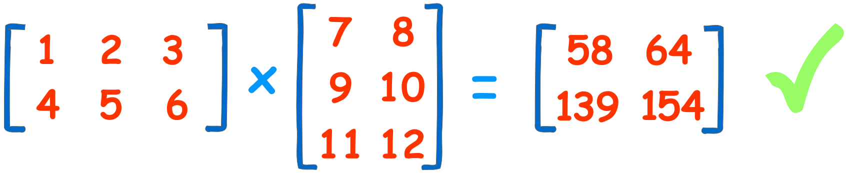
$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix} = \begin{bmatrix} 58 & \end{bmatrix}$$

1st row X 1st column:
(1, 2, 3) • (7, 9, 11) = 1×7 + 2×9 + 3×11
= 58



$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix} = \begin{bmatrix} 58 & 64 \end{bmatrix}$$

1st row X 2nd column:
(1, 2, 3) • (8, 10, 12) = 1×8 + 2×10 + 3×12
= 64



$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix} = \begin{bmatrix} 58 & 64 \\ 139 & 154 \end{bmatrix}$$ ✓

2nd row X 1st column:
(4, 5, 6) • (7, 9, 11) = 4×7 + 5×9 + 6×11
= 139

2nd row X 2nd column:
(4, 5, 6) • (8, 10, 12) = 4×8 + 5×10 + 6×12
= 154

Matrix product is possible only between matrices
nXm mXp → nXp (result dimension)

https://www.mathsisfun.com/algebra/matrix-multiplying.html

# Example

```
# Program to multiply two matrices using nested loops
# 3x3 matrix
X = [[12,7,3],
    [4 ,5,6],
    [7 ,8,9]]
# 3x4 matrix
Y = [[5,8,1,2],
    [6,7,3,0],
    [4,5,9,1]]
# result is 3x4
result = [[0,0,0,0],
         [0,0,0,0],
         [0,0,0,0]]
# iterate through rows of X
for i in range(len(X)):
   # iterate through columns of Y
   for j in range(len(Y[0])):
       # iterate through rows of Y
       for k in range(len(Y)):
           result[i][j] += X[i][k] * Y[k][j]
for r in result:
   print(r)
```

**Output:**

```
[114, 160, 60, 27]
[74, 97, 73, 14]
[119, 157, 112, 23]
```

# Example Matrix Multiplication Using Nested List Comprehension

```
# Program to multiply two matrices using list comprehension

# 3x3 matrix
X = [[12,7,3],
    [4 ,5,6],
    [7 ,8,9]]

# 3x4 matrix
Y = [[5,8,1,2],
    [6,7,3,0],
    [4,5,9,1]]

# result is 3x4
result = [[sum(a*b for a,b in zip(X_row,Y_col)) for Y_col in zip(*Y)] for X_row in X]

for r in result:
   print(r)
```

**Output:**

```
[114, 160, 60, 27]
[74, 97, 73, 14]
[119, 157, 112, 23]
```

# Exercise

Write a python script to multiply two matrices.
You can use the previous example.
The matrices can be defined inside the program or read by file.
Try the case in which matrices are in two different files or in one unique file.

Try also the special case of product between matrix and vector [mXn X nX1]

Verify with an example that
AXB != BXA [must be mXn * nXm]
Suggestion: incapsulate the matrix product in a function receiving the two matrices as parameters.