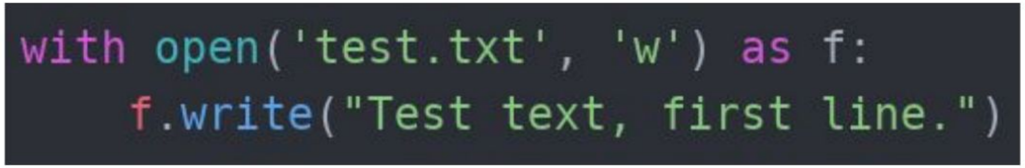
Introduction to programming with Python

context manager

context manager

The with statement defines a code block with methods defined by the context manager (an object that defines the runtime context to be established when executing the code block with). Typical uses of context managers include saving and restoring various types of global state, locking and unlocking resources, closing open files, etc

With with, the file closes automatically after the execution of the code block.



```python
with open('test.txt', 'w') as f:
    f.write("Test text, first line.")
```

The with statement ensures the file is closed regardless of potential errors within the code block. If an exception occurs before the end of the block, Python will close the file and then raise the exception.

modules

python - modules -

Modules are libraries of code that can be imported into your script or an interactive instance of the interpreter. To import a module, you use the keyword import

```
>>> import math
>>> math.factorial(5)
120
```

import the math module

```
>>> from math import factorial
>>> factorial(5)
120
```

import only a function of the module math

```
>>> from string import ascii_lowercase
>>> ascii_lowercase
'abcdefghijklmnopqrstuvwxyz'
```

import an attribute of the module string

python - function/method -

A method is a function that is part of a class and can be used on an object of the class itself. A function is an instance object of the function class.

```
>>> def function_name():
...             pass
...
>>> type(function_name)
<class 'function'>
>>> type(print)
<class 'builtin_function_or_method'>
>>>type(list().append)
<class 'builtin_function_or_method'>
>>>
```

define a function

type class 'function'

built-in function print()

Integrated method of the class list (the function is tied to the specific object)

SYNTAX

Function:
- function()

Method:
- object.method()

python - modules -

Small list of more common python modules.

pillow→ image manipulation;

matplotlib → used to create bi-dimensional mathematical graphs;

numpy → popular module for working with array, matrix, etc.;

opencv → open source computer vision module;

requests → mostly used by web developers, allows to handle HTTP requests;

sqlalchemy → module for database abstraction;

beautifulsoup → module for HTML/XML document parsing;

pandas → data scientist favourite module;

… and many more: cirq, pytorch, delorean, theano, tensorflow, keras, etc.

data structure

python - data structure -

Main Python data structure are:

- List
- Tuple
- Dict
- Set

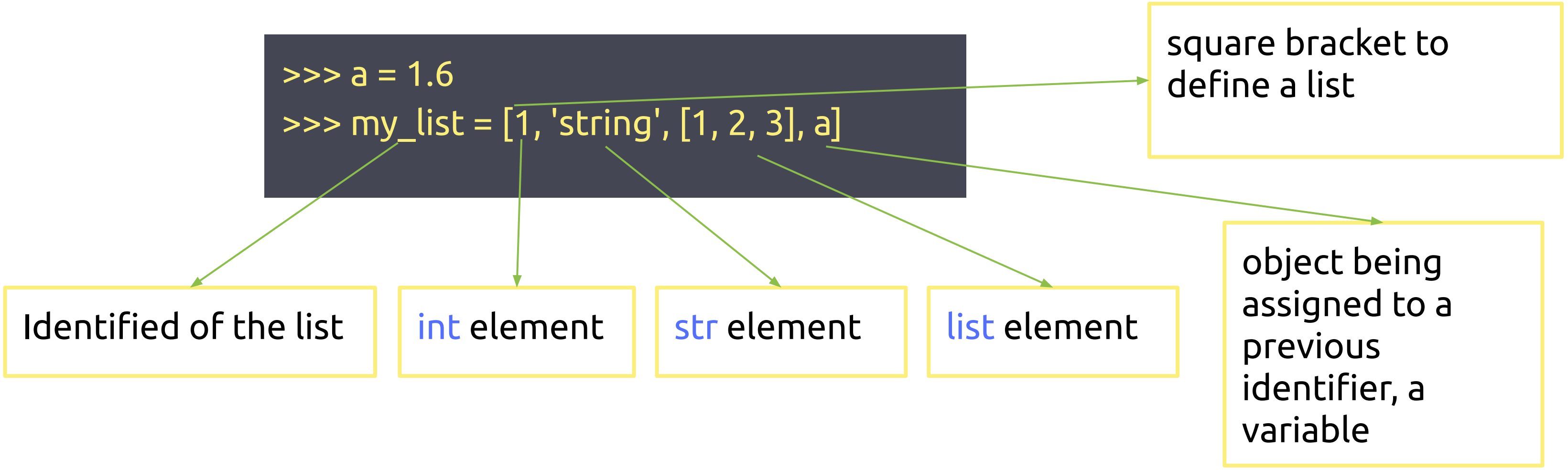
Constructors of list, tuple, dict and set

```
>>> list()
[]
>>> tuple()
()
>>> dict()
{}
>>> set()
set()
```

Additionally, there are many other data structures that can be imported from modules.

python - list -

The most versatile data structure is the list, which can be written as a list of objects (elements), separated by commas, enclosed in square brackets (or with the list() constructor), and can contain objects of different types.



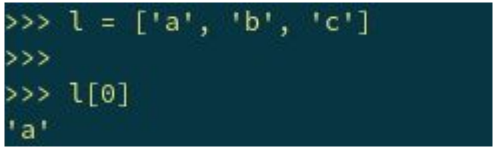
```
>>> a = 1.6
>>> my_list = [1, 'string', [1, 2, 3], a]
```

square bracket to define a list
Identified of the list
int element
str element
list element
object being assigned to a previous identifier, a variable

python - list -

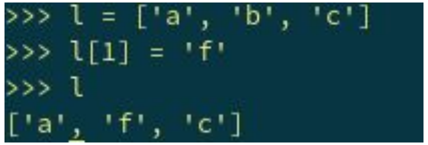
Lists are mutable collections of ordered objects that can be accessed by their index.
The index numbering starts from 0.

access to an element



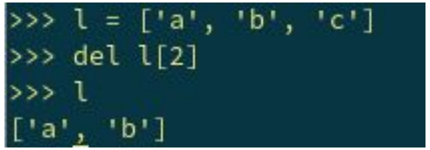
```
>>> l = ['a', 'b', 'c']
>>>
>>> l[0]
'a'
```

substitution of an element



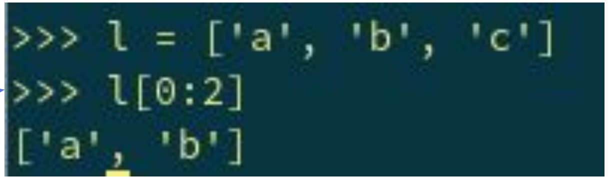
```
>>> l = ['a', 'b', 'c']
>>> l[1] = 'f'
>>> l
['a', 'f', 'c']
```

removing an element



```
>>> l = ['a', 'b', 'c']
>>> del l[2]
>>> l
['a', 'b']
```

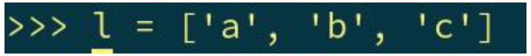
On lists (as on strings), you can also use "slicing," accessing a "slice" of elements by their index.



```
>>> l = ['a', 'b', 'c']
>>> l[0:2]
['a', 'b']
```

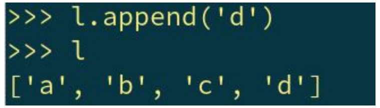
python - list/methods -

Python has a variety of built-in methods that can be used with lists. Among the most commonly used ones are:



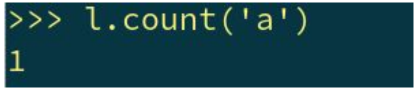
```
>>> l = ['a', 'b', 'c']
```

.append()



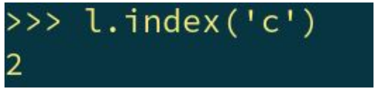
```
>>> l.append('d')
>>> l
['a', 'b', 'c', 'd']
```

.count()



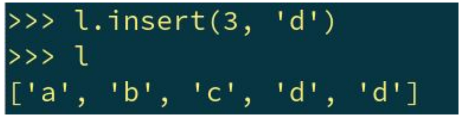
```
>>> l.count('a')
1
```

.index()



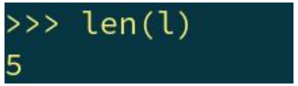
```
>>> l.index('c')
2
```

.insert()



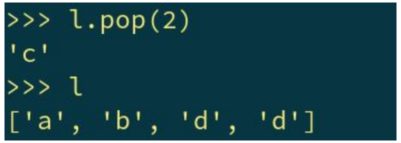
```
>>> l.insert(3, 'd')
>>> l
['a', 'b', 'c', 'd', 'd']
```

.len()



```
>>> len(l)
5
```

.pop()



```
>>> l.pop(2)
'c'
>>> l
['a', 'b', 'd', 'd']
```

python - tuple -

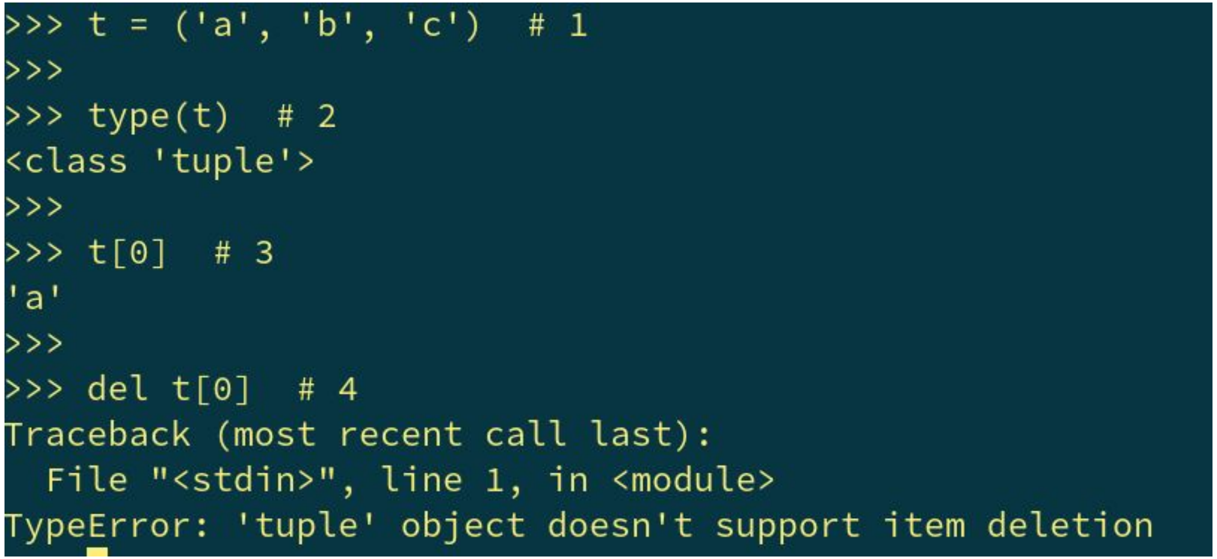
Tuples are sequences, just like lists. The main difference between tuples and lists is that tuples cannot be modified (IMMUTABLE). To instantiate a tuple, you use parentheses or the tuple() constructor.

#1 Instance of a tuple

#2 checking object type

#3 access element with index 0

#4 Error (tuple is **NOT** modifiable!)

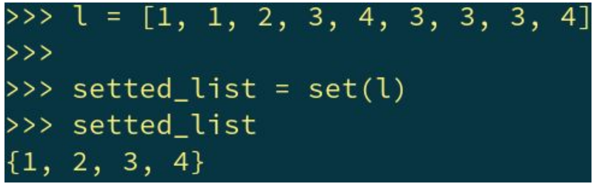


```
>>> t = ('a', 'b', 'c')  # 1
>>>
>>> type(t)  # 2
<class 'tuple'>
>>>
>>> t[0]  # 3
'a'
>>>
>>> del t[0]  # 4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object doesn't support item deletion
```

python - set -

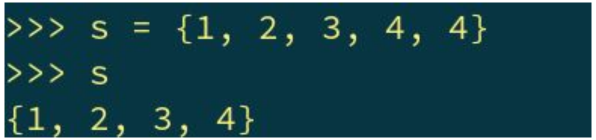
A set is an unordered and unindexed collection of unique and immutable elements. To instantiate a set, you use curly braces or the set() constructor.
The idea of a set is to recreate the typical functionality of mathematical sets.

Using set() constructor to obtain an collection of unique elements



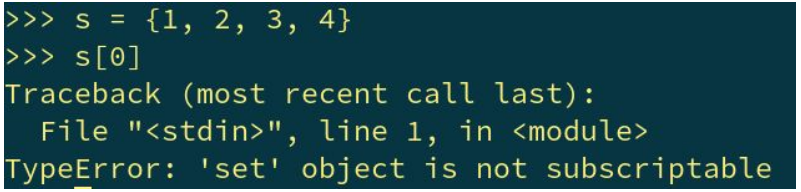
```
>>> l = [1, 1, 2, 3, 4, 3, 3, 3, 4]
>>>
>>> setted_list = set(l)
>>> setted_list
{1, 2, 3, 4}
```

Creating a set with 2 equal elements → the second "4" is not inserted.



```
>>> s = {1, 2, 3, 4, 4}
>>> s
{1, 2, 3, 4}
```

set are **NOT** indexed!



```
>>> s = {1, 2, 3, 4}
>>> s[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'set' object is not subscriptable
```

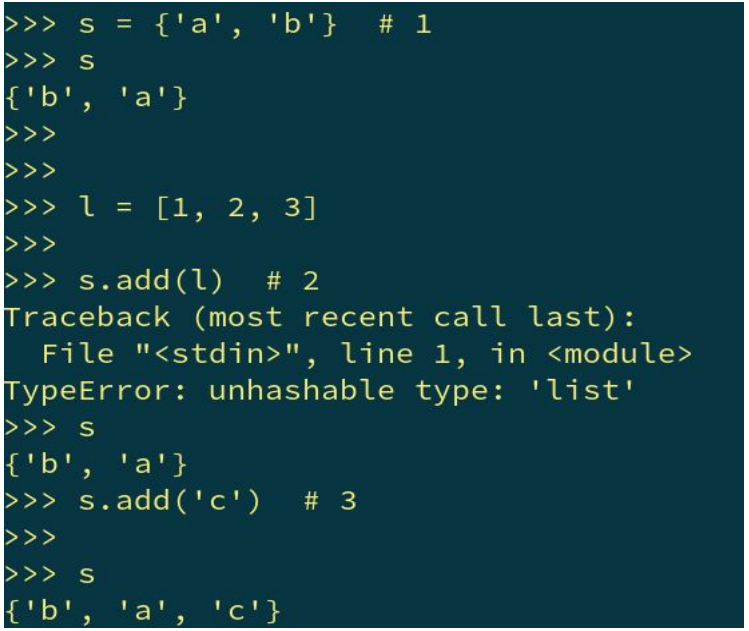
python - set/methods -

Sets have various functions and methods available for controlling the structure, modifying it, or comparing multiple structures.

#1 instance of a set

#2 I try to add a non-immutable element with .add() method, it obviously lead to an error

#3 adding an immutable (str)

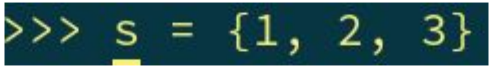


```
>>> s = {'a', 'b'}  # 1
>>> s
{'b', 'a'}
>>>
>>>
>>> l = [1, 2, 3]
>>>
>>> s.add(l)  # 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
>>> s
{'b', 'a'}
>>> s.add('c')  # 3
>>>
>>> s
{'b', 'a', 'c'}
```

Set are “**MUTABLE**” sequence but thay allow **ONLY** immutable elements (object with the implementation of the **hash** method)

python - set/methods -

Sets have the same functions used on lists (min(), max(), len()) and many methods. For the structural modification of a set, there are:

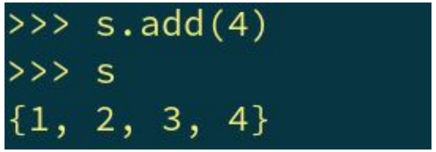


```
>>> s = {1, 2, 3}
```

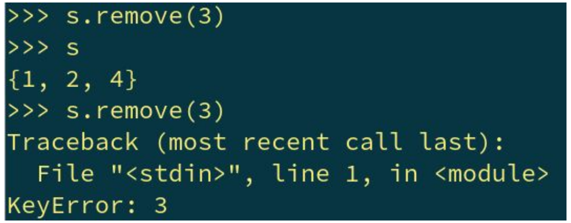
.add()

.remove()

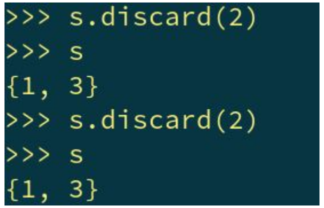
.discard()



```
>>> s.add(4)
>>> s
{1, 2, 3, 4}
```



```
>>> s.remove(3)
>>> s
{1, 2, 4}
>>> s.remove(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 3
```



```
>>> s.discard(2)
>>> s
{1, 3}
>>> s.discard(2)
>>> s
{1, 3}
```

.remove() and .discard() do the same but .remove() raise an exception if the element doesn’t exist

python - set/methods -

Other useful methods for comparing two or more sets:

Metodo	Descrizione
.difference()	Returns a set containing the differences between two or more sets.
.difference_update()	Removes the elements in the set that are present in another specified set.
.intersection()	Returns a set that is the intersection of two other sets.
.intersection_update()	Removes the elements in the set that are not present in the other/specified sets.
.isdisjoint()	Returns a boolean indicating whether two sets are empty or not of an intersection.

python - set/methods -

Other useful methods for comparing two or more sets:

Metodo	Descrizione
.issubset()	Returns a boolean indicating whether the set is included in another specified set or not.
.issuperset()	Returns a boolean indicating whether the set contains another specified set or not.
.symmetric_difference()	Returns a set with the symmetric differences of two sets.
.symmetric_difference_update()	Removes the common elements between two sets and inserts the symmetric differences into the first set.
.union()	Returns a set containing the union of one or more sets.
.update()	Modifies the set with the union of one or more sets.

python - dict -

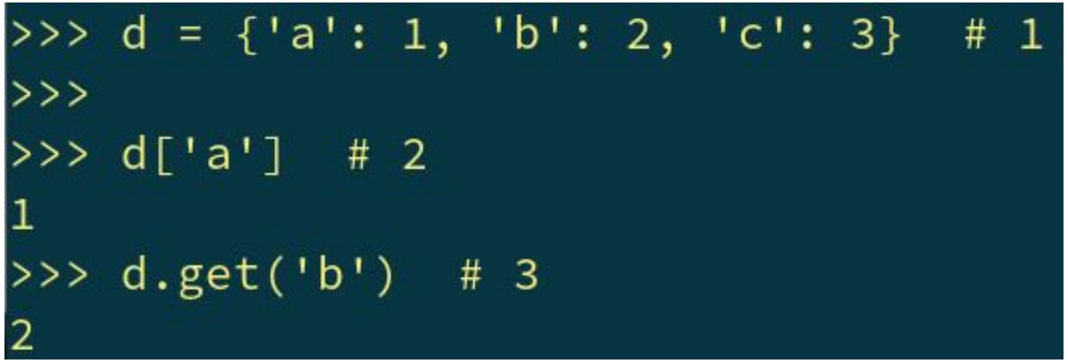
Dictionaries are sometimes found in other languages as associative memories or associative arrays. Unlike sequences (str, list, tuple), which are indexed with numbers, dictionaries are indexed by keys, which can be of any immutable type, paired with values. Strings and numbers can always be keys. They can be instantiated with curly braces or with the dict() constructor.

SYNTAX dictionary = {key: value, key: value[...]}

#1 instance of a class 'dict'

#2 access to element with "key" 'a';

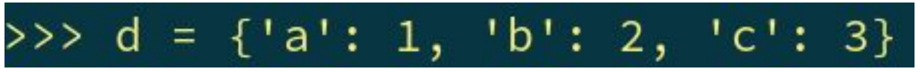
#3 access to element with "key" 'b' using .get() method



```
>>> d = {'a': 1, 'b': 2, 'c': 3}   # 1
>>>
>>> d['a']   # 2
1
>>> d.get('b')   # 3
2
```

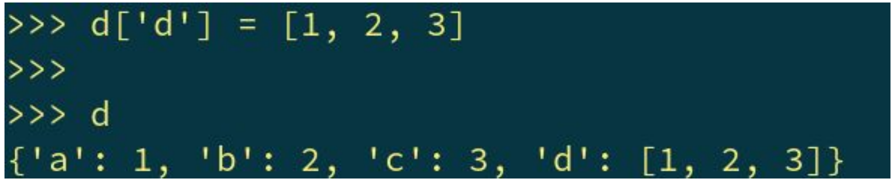
python - dizionari/funzioni metodi -

Dictionaries are mutable data structures and can contain ANY object as a value. To modify the structure of a dictionary, you can use various tools.



```
>>> d = {'a': 1, 'b': 2, 'c': 3}
```

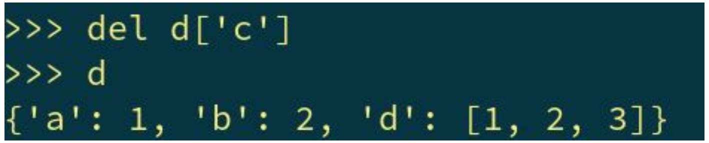
Adding a new element:
- key → ‘d’
- value → list [1, 2, 3]



```
>>> d['d'] = [1, 2, 3]
>>>
>>> d
{'a': 1, 'b': 2, 'c': 3, 'd': [1, 2, 3]}
```

Can be done also with: d.update({ ‘d’: [1, 2, 3] })

Deletion of an element with the "del" keyword (can be used with any element of mutable sequences or identifiers).



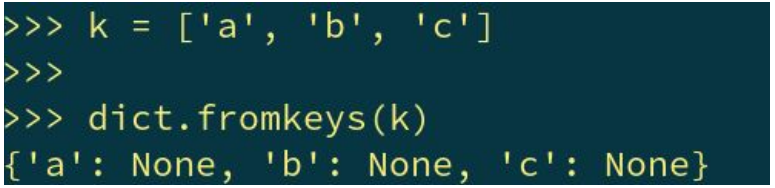
```
>>> del d['c']
>>> d
{'a': 1, 'b': 2, 'd': [1, 2, 3]}
```

python - dict/methods -

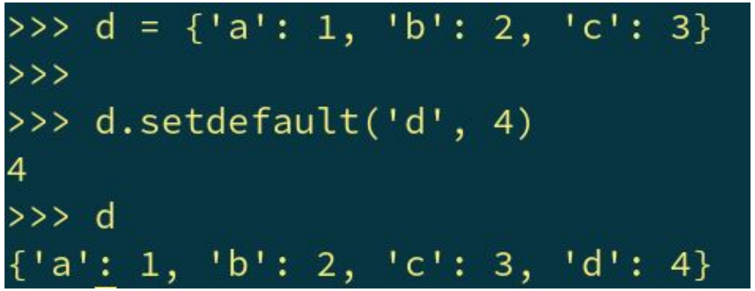
Other methods

.fromkeys(keys, [value]) → returns a dictionary with the specified keys and optional values (if no value is specified, None is assigned by default).

.setdefault(keys, [value]) → returns the value (the second argument) inserted. If the key already exists, no changes are made. If it doesn't exist, both the key and value (default=None) are inserted.



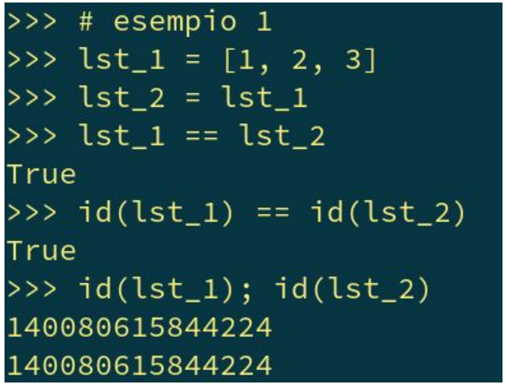
```
>>> k = ['a', 'b', 'c']
>>>
>>> dict.fromkeys(k)
{'a': None, 'b': None, 'c': None}
```



```
>>> d = {'a': 1, 'b': 2, 'c': 3}
>>>
>>> d.setdefault('d', 4)
4
>>> d
{'a': 1, 'b': 2, 'c': 3, 'd': 4}
```

python - shallow copy / deep copy -

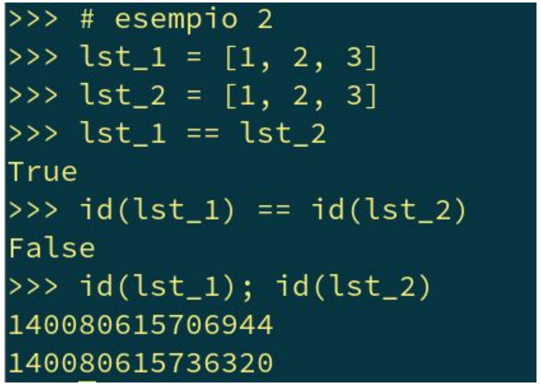
There are two types of copying: shallow copy and deep copy (which can be used by importing the "copy" module). The difference between shallow copy and deep copy is only relevant for objects with nested objects or class instances. But first, here's how Python handles "variables." [id() is a built-in function that returns the object's memory address.]



```
>>> # esempio 1
>>> lst_1 = [1, 2, 3]
>>> lst_2 = lst_1
>>> lst_1 == lst_2
True
>>> id(lst_1) == id(lst_2)
True
>>> id(lst_1); id(lst_2)
140080615844224
140080615844224
```

# example 1
- list assigned to 'lst_1'
- 'lst_1' assigned to 'lst_2'
- 'lst_1' and 'lst_2' are two identifiers of the same object

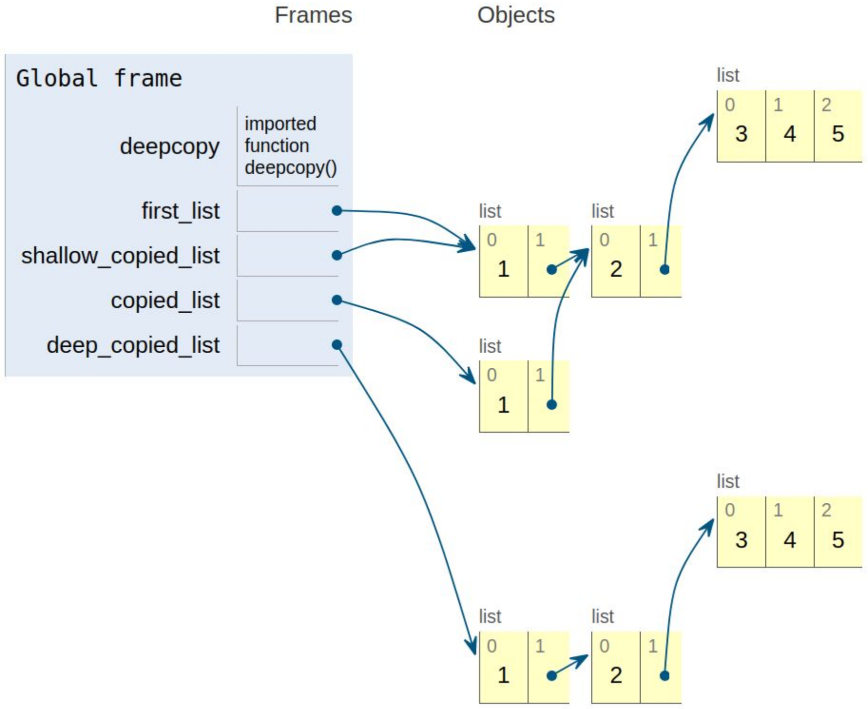
# example 2
- list assigned to 'lst_1'
- list assigned to 'lst_2'
- 'lst_1' and 'lst_2' are two identifiers that refers to different objects



```
>>> # esempio 2
>>> lst_1 = [1, 2, 3]
>>> lst_2 = [1, 2, 3]
>>> lst_1 == lst_2
True
>>> id(lst_1) == id(lst_2)
False
>>> id(lst_1); id(lst_2)
140080615706944
140080615736320
```

python - shallow copy / deep copy -

```
>>> from copy impor deepcopy
>>> first_list = [1, [2, [3, 4, 5]]]
>>> shallow_copied_list = first_list
>>> copied_list = first_list.copy()
>>> deep_copied_list = deepcopy(first_list)
>>>
```



https://pythontutor.com/