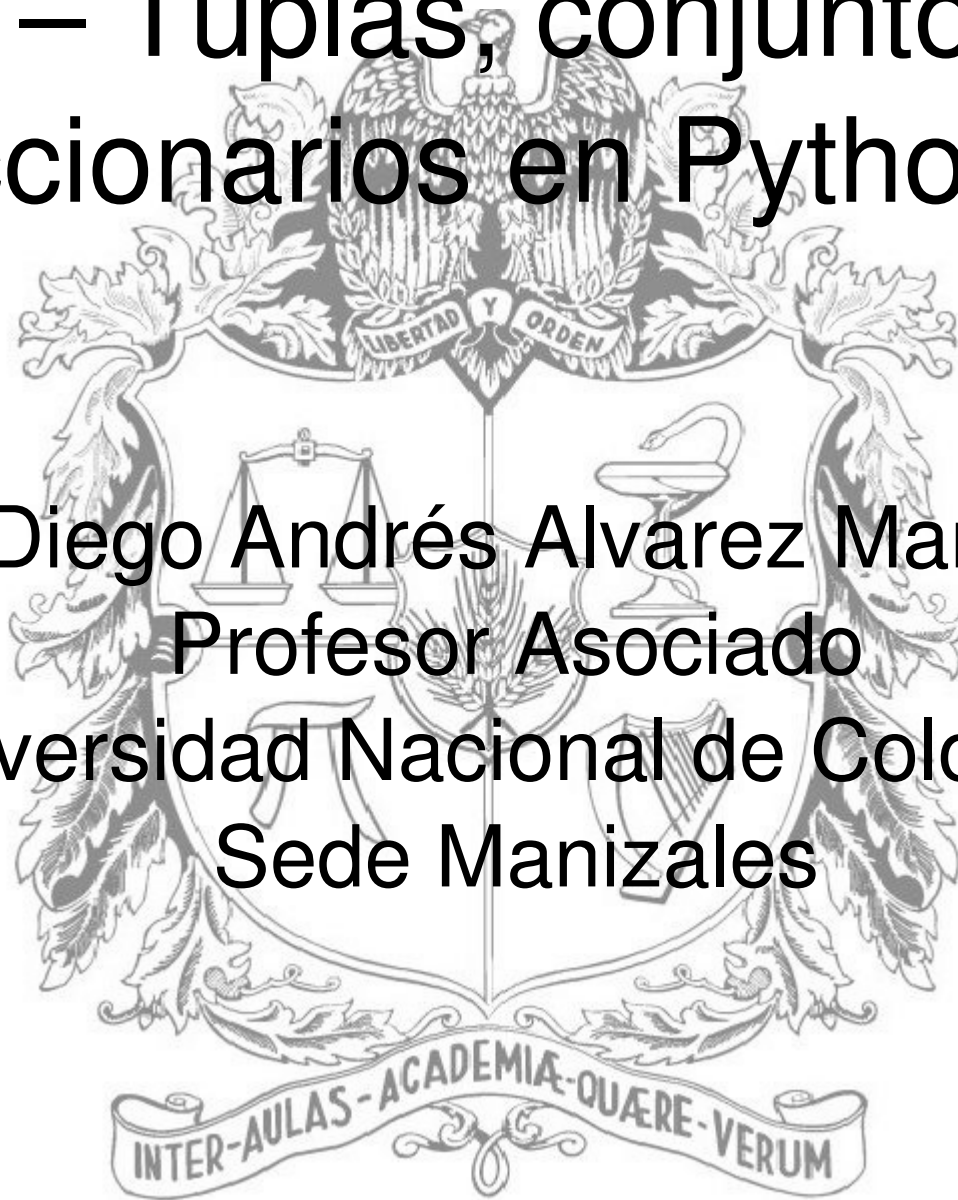


# 11 – Tuplas, conjuntos y diccionarios en Python 3

Diego Andrés Álvarez Marín  
Profesor Asociado

Universidad Nacional de Colombia  
Sede Manizales



# Tipos de datos

- Tipos de datos escalares:
  - Números enteros, flotantes, complejos, fracciones, lógicos(booleanos)
- Tipos de datos secuenciales:
  - Secuencias de bytes, cadenas
- Tipos de datos estructurados:
  - Listas (lists): secuencias ordenadas de valores
  - Tuplas (tuples): secuencias inmutables de valores ordenados
  - Conjuntos (sets): conjunto no ordenado de valores
  - Diccionarios (dictionaries): conjunto no ordenado de valores, que tienen una “llave” que los identifican
- Objetos: módulos, funciones, clases, métodos, archivos, código compilado, etc.
- “Constantes”

# Tuplas (tuples)

Una tupla, en matemáticas, es una secuencia ordenada de objetos, esto es, una lista con un número limitado de objetos. Las tuplas se emplean para describir objetos matemáticos que tienen estructura, es decir que son capaces de ser descompuestos en un cierto número de componentes. Por ejemplo:  $(x,y,z)$  se puede considerar como una tupla.

En Python, una tupla es una lista inmutable, es decir no puede cambiarse después de haberse creado.

# Creación de las tuplas

```
>>> t1 = 123, 456, 'xyz', [1, 4, -2]
>>> t1
(123, 456, 'xyz', [1, 4, -2])
>>> type(t1)
<class 'tuple'>
>>> t2 = ("a2", "b", 123, [3, 2, -1], 'xyz')
>>> t2
('a2', 'b', 123, [3, 2, -1], 'xyz')
>>> type(t2)
<class 'tuple'>
>>> t1[0]
123
>>> t1[-1]
[1, 4, -2]
>>> t2[1:3]
('b', 123)
>>> t3 = t1, t2
>>> t3
((123, 456, 'xyz', [1, 4, -2]), ('a2', 'b', 123, [3, 2, -1], 'xyz'))
>>> t1[1] = "cambio"
Traceback (most recent call last):
  File "<pyshell#21>", line 1, in <module>
    t1[1] = "cambio"
TypeError: 'tuple' object does not support item assignment
>>> |
```

Las tuplas se pueden anidar

Las tuplas son inmutables

# Tuplas vacías y tuplas con un solo elemento

```
>>> r1 = ()
>>> r1
()
>>> type(r1)
<class 'tuple'>
>>> len(r1)
0
>>> r2 = ("tupla con un solo elemento", )
>>> r2
('tupla con un solo elemento',)
>>> type(r2)
<class 'tuple'>
>>> len(r2)
1
>>> |
```

# Tuplas

Observe que las tuplas se crean y se indexan igual que las listas. La diferencia principal radica en que estas no se pueden cambiar. Las listas tienen los métodos `append()`, `extend()`, `insert()`, `remove()`, `pop()`. Las tuplas no tienen estos métodos.

Las tuplas son más rápidas que las listas. Se utilizan para definir “listas constantes”. Con ello se pueden prevenir errores.

```
>>> a_tuple  
('a', 'b', 'mpilgrim', 'z', 'example')
```

```
>>> a_tuple.append("new") ①
```

```
Traceback (innermost last):
```

```
  File "<interactive input>", line 1, in ?
```

```
AttributeError: 'tuple' object has no attribute 'append'
```

```
>>> a_tuple.remove("z") ②
```

```
Traceback (innermost last):
```

```
  File "<interactive input>", line 1, in ?
```

```
AttributeError: 'tuple' object has no attribute 'remove'
```

```
>>> a_tuple.index("example") ③
```

```
4
```

```
>>> "z" in a_tuple ④
```

```
True
```

# Asignación de elementos usando tuplas (sequence unpacking)

```
>>> L = ('xxx', 2, [1, 5, 7], False)
```

```
>>> (a,b,c,d) = L
```

```
>>> a
```

```
'xxx'
```

```
>>> b
```

```
2
```

```
>>> c
```

```
[1, 5, 7]
```

```
>>> d
```

```
False
```

```
>>> (Lun, Mar, Mie, Jue, Vie, Sab, Dom) = range(7)
```

```
>>> Lun
```

```
0
```


```
>>> Vie
```

```
4
```

```
>>> Dom
```

```
6
```

```
>>> |
```



Se puede utilizar este truco con funciones, de modo que estas retornen varios valores a la vez. La función debe retornar una tupla.



# Asignación de elementos usando tuplas (sequence unpacking)

```
>>> t = 123, 'xyz', ['a', 'b', 'c']
```

```
>>> t
```

```
(123, 'xyz', ['a', 'b', 'c'])
```

```
>>> v1, v2, v3 = t
```

```
>>> v1
```

```
123
```

```
>>> v2
```

```
'xyz'
```

```
>>> v3
```

```
['a', 'b', 'c']
```

```
>>>
```

```
>>> a = 1
```

```
>>> b = 2
```

```
>>> b, a = a, a+b
```

```
>>> a
```

```
3
```

```
>>> b
```

```
1
```

Asignación múltiple =  
tuple packing + sequence unpacking

# Tuplas

```
>>> L = [1, 2, 3, ['x', 1], 'xyz']
>>> type(L)
<class 'list'>
>>> T = tuple(L)
>>> T
(1, 2, 3, ['x', 1], 'xyz')
>>> type(T)
<class 'tuple'>
>>> list(T)
[1, 2, 3, ['x', 1], 'xyz']
>>> type(_)
<class 'list'>
>>> |
```

Las tuplas se pueden convertir en listas y viceversa utilizando las funciones **tuple()** y **list()**.

# Las listas dentro de las tuplas son mutables

```
>>> T = (1,2,3,[1,2,3])
```

```
>>> T
```

```
(1, 2, 3, [1, 2, 3])
```

```
>>> T[3] = 1
```

```
Traceback (most recent call last):
```

```
  File "<pyshell#2>", line 1, in <module>
```

```
    T[3] = 1
```

```
TypeError: 'tuple' object does not support item assignment
```

```
>>> T[3][1] = 'x'
```

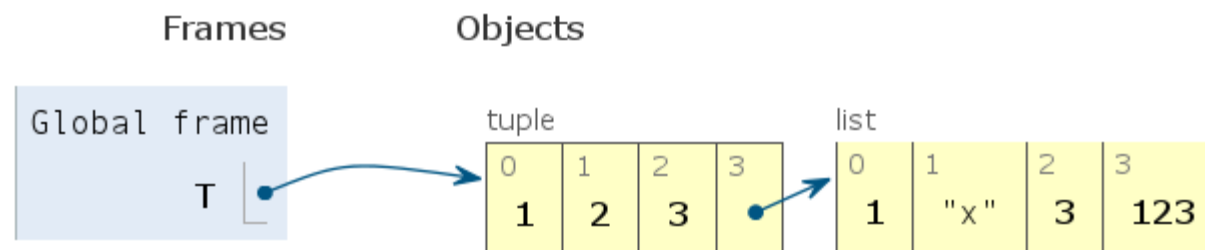
```
>>> T
```

```
(1, 2, 3, [1, 'x', 3])
```

```
>>> T[3].append(123)
```

```
>>> T
```

```
(1, 2, 3, [1, 'x', 3, 123])
```



# Comparación de secuencias

El ordenamiento se hace de igual forma como se hacía con listas, utilizando el ordenamiento lexicográfico.

```
>>> (1, 2, 3) < (1, 2, 4)
True
>>> [1, 2, 3] < [1, 2, 4]
True
>>> (1, 20, 3) < (1, 2, 4)
False
>>> [1, 20, 3] < [1, 2, 4]
False
>>> 'ABC' < 'C' < 'Pascal' < 'Python'
True
>>> 'ABC' < 'Zero' < 'Pascal' < 'Python'
False
>>> (1, 2, 3, 4) < (1, 2, 4)
True
>>> (1, 2, 3, 8) < (1, 2, 4)
True
>>> (1, 2, 30, 8) < (1, 2, 4)
False
>>> (1, 2, 3) < (1, 2, 4, 5)
True
>>> (1, 2) < (1, 2, -1)
True
>>> (1, 2, 3) == (1.0, 2.0, 3.0)
True
>>> (1, 2, ('aa', 'ab')) < (1, 2, ('abc', 'a'), 4)
True
```

# Tuplas en un contexto booleano

```
1 def es_verdadero(x):
2     if x:
3         print(x, "es verdadero")
4     else:
5         print(x, "es falso")
6
7 es_verdadero((1, 2, 'x', []))
8 es_verdadero(())
9 es_verdadero(False)
10 print(type(False))
11
12 # Observe como se crea una tupla con un solo elemento
13 es_verdadero(False, )
14 print(type(False, ))
```

Line: 18 of 36 Col: 1      LINE    INS

daalvarez@eredron:~ > python3 02\_verdadero\_falso.py

(1, 2, 'x', []) es verdadero

() es falso

False es falso

<class 'bool'>

(False,) es verdadero

<class 'tuple'>

Una tupla vacía retorna falso,  
una tupla con al menos un  
elemento retorna verdadero

# Los operadores **in** y **not in**

```
def ask_ok(prompt, retries=4, complaint='Yes or no, please!'):
    while True:
        ok = input(prompt)
        if ok in ('y', 'ye', 'yes'):
            return True
        if ok in ('n', 'no', 'nop', 'nope'):
            return False
        retries = retries - 1
        if retries < 0:
            raise OSError('uncooperative user')
    print(complaint)
```

# Tuplas vs Listas

```
>>> dir(tuple)
```

```
['__add__', '__class__', '__contains__', '__delattr__',  
 '__dir__', '__doc__', '__eq__', '__format__', '__ge__',  
 '__getattr__', '__getitem__', '__getnewargs__',  
 '__gt__', '__hash__', '__init__', '__iter__', '__le__',  
 '__len__', '__lt__', '__mul__', '__ne__', '__new__',  
 '__reduce__', '__reduce_ex__', '__repr__', '__rmul__',  
 '__setattr__', '__sizeof__', '__str__', '__subclasshook__',  
 'count', 'index']
```

```
>>> dir(list)
```

```
['__add__', '__class__', '__contains__', '__delattr__',  
 '__delitem__', '__dir__', '__doc__', '__eq__', '__format__',  
 '__ge__', '__getattr__', '__getitem__', '__gt__',  
 '__hash__', '__iadd__', '__imul__', '__init__', '__iter__',  
 '__le__', '__len__', '__lt__', '__mul__', '__ne__',  
 '__new__', '__reduce__', '__reduce_ex__', '__repr__',  
 '__reversed__', '__rmul__', '__setattr__', '__setitem__',  
 '__sizeof__', '__str__', '__subclasshook__', 'append',  
 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop',  
 'remove', 'reverse', 'sort']
```

# Conjuntos (sets)

Son colecciones de elementos no duplicados. Los conjuntos están formados por valores inmutables, sin embargo los conjuntos como tal son mutables. Con ellos se pueden realizar las operaciones de unión, intersección y diferencia.

```
>>> un_conjunto = {1, 2, 2, 1, 2, 3, 2}
>>> un_conjunto
{1, 2, 3}
>>> type(un_conjunto)
<class 'set'>
>>> una_lista = ['a', 'b', 1, 'a', 2, 'a']
>>> una_lista
['a', 'b', 1, 'a', 2, 'a']
>>> un_conjunto = set(una_lista)
>>> un_conjunto
{'b', 1, 2, 'a'}
```

Los elementos no necesariamente quedan ordenados.



# Conjuntos (sets)

```
>>> un_conjunto = {1, 2, 2, 1, 2, 3, 2}
>>> un_conjunto
{1, 2, 3}
>>> type(un_conjunto)
<class 'set'>
>>> una_lista = ['a', 'b', 1, 'a', 2, 'a']
>>> una_lista
['a', 'b', 1, 'a', 2, 'a']
>>> un_conjunto = set(una_lista)
>>> un_conjunto
{'b', 1, 2, 'a'}
>>> un_conjunto = set()
>>> un_conjunto
set()
>>> type(un_conjunto)
<class 'set'>
>>> len(un_conjunto)
0
>>> xxx = {}
>>> type(xxx)
<class 'dict'>
```

```
>>> {'casa'}
{'casa'}
>>> set('casa')
{'a', 's', 'c'}
```

Así se crea un conjunto vacío

Esto es un diccionario vacío

## Los conjuntos no se pueden indexar

```
>>> conj = {1,1,2,3,4}
>>> conj
{1, 2, 3, 4}
>>> conj[2]
Traceback (most recent call last):
  File "<pyshell#18>", line 1, in <module>
    conj[2]
TypeError: 'set' object does not support indexing
```

## Los elementos de un conjunto no pueden ser mutables

```
>>> {[1,2,3]}
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    {[1,2,3]}
TypeError: unhashable type: 'list'
>>> d = {'a':1, 'b':2}
>>> {d}
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    {d}
TypeError: unhashable type: 'dict'
```

# Agregando elementos a un conjunto

```
>>> S = {1, 2}
>>> S.add(4)
>>> S
{1, 2, 4}
>>> S.add([4])
Traceback (most recent call last):
  File "<pyshell#27>", line 1, in <module>
    S.add([4])
TypeError: unhashable type: 'list'
>>> S.add(1)
>>> S
{1, 2, 4}
>>> len(S)
3
>>> S.update(5)
Traceback (most recent call last):
  File "<pyshell#31>", line 1, in <module>
    S.update(5)
TypeError: 'int' object is not iterable
>>> S.update({5, 6, 7})
>>> S
{1, 2, 4, 5, 6, 7}
>>> S.update({3, 6, 7}, {2, 5, 8})
>>> S
{1, 2, 3, 4, 5, 6, 7, 8}
>>> S.update([7, 9, 1, 2])
>>> S
{1, 2, 3, 4, 5, 6, 7, 8, 9}
```

`add()` agrega un solo elemento

`update()` agrega o un conjunto, o varios conjuntos o una lista de elementos al conjunto original.

Los elementos repetidos no se tienen en cuenta

# Removiendo elementos de un conjunto

```
>>> S  
{1, 2, 3, 4, 5, 6, 7, 8, 9}
```

```
>>> S.discard(5)
```

```
>>> S  
{1, 2, 3, 4, 6, 7, 8, 9}
```

```
>>> S.discard(5)
```

```
>>> S  
{1, 2, 3, 4, 6, 7, 8, 9}
```

```
>>> S.remove(4)
```

```
>>> S  
{1, 2, 3, 6, 7, 8, 9}
```

```
>>> S.remove(4)
```

```
Traceback (most recent call last):
```

```
  File "<pyshell#50>", line 1, in <module>
```

```
    S.remove(4)
```

```
KeyError: 4
```

# Removiendo elementos de un conjunto

```
>>> help(set.pop)
Help on method_descriptor:
```

```
pop(...)
    Remove and return an arbitrary set element.
    Raises KeyError if the set is empty.
```

```
>>> S
{2, 3, 6, 7, 8, 9}
```

```
>>> S.pop()
```

```
2
```

```
>>> S
{3, 6, 7, 8, 9}
```

```
>>> S.clear()
```

```
>>> S
```

```
set()
```

```
>>> S.pop()
```

```
Traceback (most recent call last):
```

```
File "<pyshell#57>", line 1, in <module>
```

```
S.pop()
```

```
KeyError: 'pop from an empty set'
```

OJO: se supone que pop() saca un elemento cualquiera del conjunto. Sin embargo en mis pruebas siempre saca el elemento con "índice 0" (Python 3.4.2.)

Si el conjunto está vacío, pop() lanza la excepción KeyError.

# Operaciones con conjuntos

```
>>> a = {1, 3, 5, 1, -2, 'xyz', [1,2,3], 'xyz'}
```

```
Traceback (most recent call last):
```

```
  File "<pyshell#83>", line 1, in <module>
```

```
    a = {1, 3, 5, 1, -2, 'xyz', [1,2,3], 'xyz'}
```

```
TypeError: unhashable type: 'list'
```

```
>>> a = {1, 3, 5, 1, -2, 'xyz', 'xyz'}
```

```
>>> a
```

```
{1, 3, 'xyz', 5, -2}
```

```
>>> b = {9, 7, 5, 3, 1, 1}
```

```
>>> b
```

```
{1, 9, 3, 5, 7}
```

```
>>> 'xyz' in a
```

```
True
```

```
>>> 'xyz' in b
```

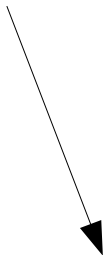
```
False
```

```
>>> letras = set('abracadabra')
```

```
>>> letras
```

```
{'c', 'r', 'a', 'd', 'b'}
```

```
>>> |
```



Recuerde que los elementos que forman el conjunto deben ser de tipo inmutable

# Operaciones con conjuntos

```
>>> a
{1, 3, 'xyz', 5, -2}
>>> b
{1, 9, 3, 5, 7}
>>> a + b                                # este comando no existe con conjuntos
Traceback (most recent call last):
  File "<pyshell#134>", line 1, in <module>
    a + b                                # este comando no existe con conjuntos
TypeError: unsupported operand type(s) for +: 'set' and 'set'
>>> a | b                                # union de conjuntos
{1, 3, 5, 7, 9, 'xyz', -2}
>>> a.union(b)
{1, 3, 5, 7, 9, 'xyz', -2}
>>> a & b                                # intersección de conjuntos
{1, 3, 5}
>>> a.intersection(b)
{1, 3, 5}
>>> a - b                                # elementos en a que no están en b
{'xyz', -2}
>>> a.difference(b)
{'xyz', -2}
>>> a ^ b                                # diferencia simétrica de conjuntos
{7, 9, 'xyz', -2}
>>> a.symmetric_difference(b)
{7, 9, 'xyz', -2}
>>> a
{1, 3, 'xyz', 5, -2}
>>> b
{1, 9, 3, 5, 7}
```

# Los conjuntos son mutables

```
>>> x = {4, -3, 2, 1, 2, 1}
>>> x
{1, 2, 4, -3}
>>> hex(id(x))
'0x7f8b07afde48'
>>> y = x
>>> y |= {'a', 'b', 'c'}
>>> y
{1, 2, 'a', 4, 'c', 'b', -3}
>>> hex(id(y))
'0x7f8b07afde48'
>>> x
{1, 2, 'a', 4, 'c', 'b', -3}
>>> x is y
True
```



# Operaciones con conjuntos

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

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

```
>>> a_set.issubset(b_set)    ①  $A \subseteq B$ 
```

```
True
```

```
>>> b_set.issuperset(a_set)  ②  $A \supseteq B$ 
```

```
True
```

```
>>> a_set.add(5)             ③
```

```
>>> a_set.issubset(b_set)
```

```
False
```

```
>>> b_set.issuperset(a_set)
```

```
False
```

# Conjuntos en un contexto booleano

```
1  def es_verdadero(x):  
2      if x:  
3          print(x, "es verdadero")  
4      else:  
5          print(x, "es falso")  
6  
7      es_verdadero({1, 2, 'x'})  
8      es_verdadero(set())  
9      es_verdadero({False})  
10
```

Line: 11 of 45 Col: 8      LINE    INS

```
daa@heimdall ~ $ python3 02_verdadero_falso.py  
{1, 2, 'x'} es verdadero  
set() es falso  
{False} es verdadero
```

Un conjunto vacío retorna falso, un conjunto con al menos un elemento retorna verdadero

# Diccionarios

## (arrays asociativos)

[http://en.wikipedia.org/wiki/Associative\\_array](http://en.wikipedia.org/wiki/Associative_array)

Conjunto de 'llave': valor

```
>>> D = {'nombre': 'Pepito', 'Apellido': 'Pérez', 'edad': 20}
>>> D
{'Apellido': 'Pérez', 'nombre': 'Pepito', 'edad': 20}
>>> D['nombre']
'Pepito'
>>> D['edad']
20
>>> D['Pepito']
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    D['Pepito']
KeyError: 'Pepito'
>>> |
```

# Diccionarios

```
>>> D
{'Apellido': 'Pérez', 'nombre': 'Pepito', 'edad': 20}
>>> D['nombre'] = 'Jorge'
>>> D
{'Apellido': 'Pérez', 'nombre': 'Jorge', 'edad': 20}
>>> D['edad'] = 26
>>> D
{'Apellido': 'Pérez', 'nombre': 'Jorge', 'edad': 26}
>>> D['Nombre'] = 'Julián'
>>> D
{'Apellido': 'Pérez', 'Nombre': 'Julián', 'nombre': 'Jorge', 'edad': 26}
>>> D['Sexo'] = 'M'
>>> D
{'Apellido': 'Pérez', 'Nombre': 'Julián', 'nombre': 'Jorge', 'edad': 26, 'Sexo': 'M'}
>>>
```

```

1  ▼ sufijos = { 1000: ['KB', 'MB', 'GB', 'TB'],
2              1024: ['KiB', 'MiB', 'GiB', 'TiB']}
3
4  ▼ def tamaño_aproximado(tamaño, exacto=True):
5  ▼     ''' Convierte un número de bytes a otras unidades
6
7         tamaño_aproximado(tamaño, exacto=True)
8
9         Parámetros de entrada:
10             tamaño: tamaño dado en bytes
11             exacto: True  = división entre 1024
12                   False = división entre 1000
13
14         Retorna una cadena de texto
15
16     '''
17  ▼ if tamaño < 0:
18     raise ValueError('"tamaño" debe ser positivo')
19
20     multiple = 1024 if exacto else 1000
21  ▼ for s in sufijos[multiple]:
22     tamaño /= multiple
23  ▼     if tamaño < multiple:
24         return '{0:.1f} {1}'.format(tamaño, s)
25
26     raise ValueError('"tamaño" es demasiado grande')
27
28  ▼ if __name__ == '__main__':
29     print(tamaño_aproximado(1000000000, False))
30     print(tamaño_aproximado(1000000000))

```

```

daa@heimdall ~ $ python3 01_primer_ejemplo.py
1.0 GB
953.7 MiB
daa@heimdall ~ $ █

```

# Los diccionarios son mutables

```
>>> D = {"Nombre": "Jorge", "Edad": 23, "Sexo" : "M" }
>>> D2 = D
>>> D3 = D.copy()
>>> hex(id(D))
'0x7f9b22470208'
>>> hex(id(D2))
'0x7f9b22470208'
>>> hex(id(D3))
'0x7f9b22470108'
>>> D2["Edad"] = 100
>>> D
{'Nombre': 'Jorge', 'Edad': 100, 'Sexo': 'M'}
>>> D2
{'Nombre': 'Jorge', 'Edad': 100, 'Sexo': 'M'}
>>> D3
{'Nombre': 'Jorge', 'Edad': 23, 'Sexo': 'M'}
```

Existe un método `copy()` para copiar diccionarios

# El corto circuito y el acceso a diccionarios

You can use the `in` operator together with short-circuit evaluation to avoid raising an error when trying to access a key that is not in the dictionary:

Python

```
>>> MLB_team['Toronto']
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    MLB_team['Toronto']
KeyError: 'Toronto'

>>> 'Toronto' in MLB_team and MLB_team['Toronto']
False
```

In the second case, due to short-circuit evaluation, the expression `MLB_team['Toronto']` is not evaluated, so the `KeyError` exception does not occur.

# Diccionarios

Los diccionarios deben tener como llaves tipos inmutables como números, tuplas o cadenas. Los valores pueden ser de cualquier tipo de datos.

```
>>> sufijos = { 1000: ['KB', 'MB', 'GB', 'TB'],
                1024: ['KiB', 'MiB', 'GiB', 'TiB']}

>>> sufijos
{1000: ['KB', 'MB', 'GB', 'TB'], 1024: ['KiB', 'MiB', 'GiB', 'TiB']}
>>> len(sufijos)
2
>>> 1000 in sufijos
True
>>> sufijos[1000]
['KB', 'MB', 'GB', 'TB']
>>> sufijos[1000][2]
'GB'
>>>
```

Tuples can be used as keys if they contain only strings, numbers, or tuples; if a tuple contains any mutable object either directly or indirectly, it cannot be used as a key. You can't use lists as keys, since lists can be modified in place using index assignments, slice assignments, or methods like `append()` and `extend()`.



# Las llaves (keys) del diccionario deben ser inmutables

Almost any type of value can be used as a dictionary key in Python. You just saw this example, where integer, float, and Boolean objects are used as keys:

Python

```
>>> foo = {42: 'aaa', 2.78: 'bbb', True: 'ccc'}
>>> foo
{42: 'aaa', 2.78: 'bbb', True: 'ccc'}
```

Evite utilizar floats como llaves en los diccionarios. Podría tener problemas accediendo a este valor.

You can even use built-in objects like types and functions:

Python

```
>>> d = {int: 1, float: 2, bool: 3}
>>> d
{<class 'int'>: 1, <class 'float'>: 2, <class 'bool'>: 3}
>>> d[float]
2

>>> d = {bin: 1, hex: 2, oct: 3}
>>> d[oct]
3
```

Secondly, a dictionary key must be of a type that is immutable. That means an integer, float, string, or Boolean can be a dictionary key, as you have seen above. A tuple can also be a dictionary key, because tuples are immutable:

Python

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

Recall from the discussion on [tuples](#) that one rationale for using a tuple instead of a list is that there are circumstances where an immutable type is required. This is one of them.

However, neither a list nor another dictionary can serve as a dictionary key, because lists and dictionaries are mutable:

Python

```
>>> d = {[1, 1]: 'a', [1, 2]: 'b', [2, 1]: 'c', [2, 2]: 'd'}
Traceback (most recent call last):
  File "<pyshell#20>", line 1, in <module>
    d = {[1, 1]: 'a', [1, 2]: 'b', [2, 1]: 'c', [2, 2]: 'd'}
TypeError: unhashable type: 'list'
```

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'jack': 4098, 'guido': 4127}
>>> tel['jack']
4098
>>> del tel['sape'] → Borra la pareja 'llave':valor asociado
>>> tel['irv'] = 4127
>>> tel
{'irv': 4127, 'jack': 4098, 'guido': 4127}
>>> tel.keys()
dict_keys(['irv', 'jack', 'guido'])
>>> list(tel.keys()) → Lista las etiquetas (sin ordenar – pueden
['irv', 'jack', 'guido'] aparecer en cualquier orden)
>>> sorted(tel.keys()) → Lista las etiquetas (ordenadas)
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
>>> 'jack' not in tel
False
>>>
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'jack': 4098, 'guido': 4127, 'sape': 4139}
>>>
```

# Los métodos keys() y values() de los diccionarios

```
>>> d = {'nombre': 'Pedro', 123: 'xyz', 'valor': 4532.3}
>>> d
{'nombre': 'Pedro', 'valor': 4532.3, 123: 'xyz'}
>>> d.keys()
dict_keys(['nombre', 'valor', 123])
>>> list(d.keys())
['nombre', 'valor', 123]
>>> d.values()
dict_values(['Pedro', 4532.3, 'xyz'])
>>> list(d.values())
['Pedro', 4532.3, 'xyz']
```

# Creando diccionarios con dict()

Cuando todas las llaves son cadenas, es opcional poner las comillas en las llaves.

```
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'jack': 4098, 'guido': 4127}
>>>
>>>
>>> dict(sape=4139, guido=4127, jack=4098)
{'sape': 4139, 'jack': 4098, 'guido': 4127}
```

To illustrate, the following examples all return a dictionary equal to `{"one": 1, "two": 2, "three": 3}`:

```
>>> a = dict(one=1, two=2, three=3)
>>> b = {'one': 1, 'two': 2, 'three': 3}
>>> c = dict(zip(['one', 'two', 'three'], [1, 2, 3]))
>>> d = dict([('two', 2), ('one', 1), ('three', 3)])
>>> e = dict({'three': 3, 'one': 1, 'two': 2})
>>> a == b == c == d == e
True
```

# Los diccionarios y los conjuntos no tienen orden, las listas si lo tienen

```
>>> D1 = {'animal':'gato', 'nombre':'Laura', 'valor':123.43}
>>> D2 = {'valor':123.43, 'animal':'gato', 'nombre':'Laura'}
>>> D1
{'nombre': 'Laura', 'valor': 123.43, 'animal': 'gato'}
>>> D2
{'valor': 123.43, 'nombre': 'Laura', 'animal': 'gato'}
>>> D1 == D2
True
>>> S1 = {'gato', 'Laura', 123.43}
>>> S1
{123.43, 'gato', 'Laura'}
>>> S2 = {123.43, 'gato', 'Laura'}
>>> S2
{'gato', 123.43, 'Laura'}
>>> S1 == S2
True
>>> L1 = ['gato', 'Laura', 123.43]
>>> L1
['gato', 'Laura', 123.43]
>>> L2 = [123.43, 'gato', 'Laura']
>>> L2
[123.43, 'gato', 'Laura']
>>> L1 == L2
False
```

**¡CUIDADO!** A partir de Python 3.7 los diccionarios ya son estructuras ordenadas. Sin embargo, en los foros he visto comentarios que dicen que uno no debería confiar en este comportamiento, ya que podría cambiar en el futuro de nuevo.

<https://stackoverflow.com/questions/39980323/are-dictionaries-ordered-in-python-3-6?rq=1>

# for en diccionarios

aquí-se utilizó **sorted()**  
para darle cierta  
estructura a la  
presentación de los datos  
del diccionario

```
1 d = {'xxx': 123, 'yyy': 456, 'zzz': 789}
2
3 ▼ for k in d:
4     print(k)
5     ...
6 ▼ for k in d:
7     print(d[k])
8
9 d['rrr'] = -1
10 print(d)
11
12 ▼ for k, v in d.items():
13     print(k, v)
14
15 ▼ for k, v in sorted(d.items()):
16     print(k, v)
```

Line 17, Column 1

```
daalvarez@eredron ~ $ python3 11_for_dictionaries.py
```

```
xxx
yyy
zzz
123
456
789
{'xxx': 123, 'yyy': 456, 'zzz': 789, 'rrr': -1}
xxx 123
yyy 456
zzz 789
rrr -1
rrr -1
xxx 123
yyy 456
zzz 789
```

```
daalvarez@eredron ~ $ █
```

# Diccionarios en un contexto booleano

```
1  def es_verdadero(x):
2      if x:
3          print(x, "es verdadero")
4      else:
5          print(x, "es falso")
6
7  es_verdadero({'a':1, 'b':[1, 3], 'c':'c'})
8  es_verdadero({})
```

Line: 15 of 55 Col: 18      LINE    INS

```
daa@heimdall ~ $ python3 02_verdadero_falso.py
{'a': 1, 'b': [1, 3], 'c': 'c'} es verdadero
{} es falso
```

Un diccionario vacío retorna falso, un diccionario con al menos un par llave:valor retorna verdadero



# Algunas funciones y métodos con diccionarios

```
>>> d = {'a': 10, 'b': 20, 'c': 30}
>>> len(d)
3
>>> d.get('a')
10
>>> print(d.get('hola'))
None
>>> print(d.get('hola', -1))
-1
>>> d.items()
dict_items([('c', 30), ('b', 20), ('a', 10)])
>>> list(d.items())
[('c', 30), ('b', 20), ('a', 10)]
>>> d.keys()
dict_keys(['c', 'b', 'a'])
>>> list(d.keys())
['c', 'b', 'a']
>>> d.values()
dict_values([30, 20, 10])
>>> list(d.values())
[30, 20, 10]
```

```
>>> D = {42: 'aaa', 2.78: 'bbb', True: 'ccc', "cadena": "hola"}
>>> D
{True: 'ccc', 42: 'aaa', 'cadena': 'hola', 2.78: 'bbb'}
>>> D.pop(42)
'aaa'
>>> D
{True: 'ccc', 'cadena': 'hola', 2.78: 'bbb'}
>>> D.pop("xxx")
Traceback (most recent call last):
  File "<pyshell#92>", line 1, in <module>
    D.pop("xxx")
KeyError: 'xxx'
>>> D.pop("xxx", -1)
-1
```

# Como concatenar diccionarios

```
>>> d1 = {'a': 10, 'b': 20, 'c': 30}
>>> d2 = {'b': 200, 'd': 400}
>>>
>>> d1.update(d2)
>>>
>>> d1
{'c': 30, 'd': 400, 'b': 200, 'a': 10}
>>>
>>> d1.update([('x', 100), ('y', -600)])
>>> d1
{'x': 100, 'd': 400, 'y': -600, 'a': 10, 'c': 30, 'b': 200}
```

```
>>> x = {"a":1, "b":2}
>>> y = {"b":3, "c":4}
>>> z = {**x, **y}
>>> z
{'b': 3, 'c': 4, 'a': 1}
```

→ A partir de Python 3.5 se puede utilizar este truco para concatenar diccionarios.

# Set and dict comprehensions

```
>>> # list comprehension
>>> [x for x in 'abracadabra' if x not in 'abc']
['r', 'd', 'r']
>>> # set comprehension
>>> {x for x in 'abracadabra' if x not in 'abc'}
{'r', 'd'}
>>> # dict comprehension
>>> {x : x**2 for x in (2, 4, 6)}
{2: 4, 4: 16, 6: 36}
>>> |
```

# Funciones variádicas, tuplas y diccionarios

```
1 def mi_fun(aa1, aa2, *bbb, **ccc):
2     print('El argumento aa1 es', aa1)
3     print('El argumento aa2 es', aa2)
4     print('-'*70)
5     print('El argumento bbb es', bbb)
6     for i,b in enumerate(bbb):
7         print('bbb[', i, '] =', b)
8     print('-'*70)
9     print('El argumento ccc es', ccc)
10    keys = sorted(ccc.keys())
11    for kw in keys:
12        print('ccc[', kw, '] =', ccc[kw])
13
14    mi_fun('Arg 1', 'Arg 2', 'Arg 3', 'Arg 4', a2 = 'Arg 5', a3 = 123, a1 = [1,2,3])
```

- El argumento \* va antes que el argumento \*\*
- El argumento \* recibe una tupla
- El argumento \*\* recibe un diccionario
- El orden con los que se guardan los argumentos en \*\* es indefinido

Line: 16 of 16 Col: 1      LINE    INS

```
daalvarez@eredron:~ > python3 04_parametros_variables_a_una_funcion.py
```

```
El argumento aa1 es Arg 1
```

```
El argumento aa2 es Arg 2
```

```
-----
El argumento bbb es ('Arg 3', 'Arg 4')
```

```
bbb[ 0 ] = Arg 3
```

```
bbb[ 1 ] = Arg 4
```

```
-----
El argumento ccc es {'a3': 123, 'a1': [1, 2, 3], 'a2': 'Arg 5'}
```

```
ccc[ a1 ] = [1, 2, 3]
```

```
ccc[ a2 ] = Arg 5
```

```
ccc[ a3 ] = 123
```

```
daalvarez@eredron:~ > □
```

### 4.7.3. Arbitrary Argument Lists

Finally, the least frequently used option is to specify that a function can be called with an arbitrary number of arguments. These arguments will be wrapped up in a tuple (see [Tuples and Sequences](#)). Before the variable number of arguments, zero or more normal arguments may occur.

```
def write_multiple_items(file, separator, *args):  
    file.write(separator.join(args))
```

Normally, these `variadic` arguments will be last in the list of formal parameters, because they scoop up all remaining input arguments that are passed to the function. Any formal parameters which occur after the `*args` parameter are 'keyword-only' arguments, meaning that they can only be used as keywords rather than positional arguments.

```
>>> def concat(*args, sep="/"):  
...     return sep.join(args)  
...  
>>> concat("earth", "mars", "venus")  
'earth/mars/venus'  
>>> concat("earth", "mars", "venus", sep=".")  
'earth.mars.venus'
```

## 4.7.4. Unpacking Argument Lists

The reverse situation occurs when the arguments are already in a list or tuple but need to be unpacked for a function call requiring separate positional arguments. For instance, the built-in `range()` function expects separate *start* and *stop* arguments. If they are not available separately, write the function call with the `*`-operator to unpack the arguments out of a list or tuple:

```
>>> list(range(3, 6))           # normal call with separate arguments
[3, 4, 5]
>>> args = [3, 6]
>>> list(range(*args))         # call with arguments unpacked from a list
[3, 4, 5]
```

In the same fashion, dictionaries can deliver keyword arguments with the `**`-operator:

```
>>> def parrot(voltage, state='a stiff', action='vroom'):
...     print("-- This parrot wouldn't", action, end=' ')
...     print("if you put", voltage, "volts through it.", end=' ')
...     print("E's", state, "!")
...
>>> d = {"voltage": "four million", "state": "bleedin' demised", "action": "VOOM"}
>>> parrot(**d)
-- This parrot wouldn't VOOM if you put four million volts through it. E's bleedin' demised !
```

```

1 def ordenar_listas_cadenas(L):
2     """
3     Organiza alfabéticamente una lista con cadenas de texto, teniendo en cuenta
4     que dichas cadenas pueden tener tildes, diéresis o la letra ñ.
5     :param L: lista a ordenar
6     :return: la lista ordenada
7     """
8
9     # Crea la tabla para las conversiones de caracteres
10    tabla = str.maketrans("áéíóúñÁÉÍÓÚÑ", "aeiouunAEIOUUN")
11
12    # Se crea un iterador de 0 a len(L)-1
13    range_num = range(len(L))
14
15    # Se les quita la tilde a cada una de las palabras de la lista
16    L_sin_tilde = [L[i].translate(tabla) for i in range_num]
17
18    # Se ordena la lista y junto con sus índices de ordenamiento
19    tmp = sorted(zip(L_sin_tilde, range_num))
20
21    # se retorna la lista ordenada (L_ord) y los índices después del
22    # ordenamiento (idx). Aquí se está desempacando el contenido de tmp
23    # Ver: https://docs.python.org/3/library/functions.html#zip
24    L_ord, idx = zip(*tmp)
25
26    # Se retorna la lista ordenada
27    return [L[idx[i]] for i in range_num]
28
29    lista_palabras = "murciélago árbol nunca otoño ñoño érase cigüeña".split()
30
31    # Lista de palabras mal ordenadas con sorted()
32    print(sorted(lista_palabras))
33
34    # Lista de palabras correctamente ordenadas
35    print(ordenar_listas_cadenas(lista_palabras))

```

NOTA: este algoritmo contiene un error ordenando la letra ñ. Falta corregir.

Line: 37 of 37 Col: 1

LINE INS

11\_ordenar\_cadenas.py

```

daalvarez@eredron:~ > python3 11_ordenar_cadenas.py
['cigüeña', 'murciélago', 'nunca', 'otoño', 'árbol', 'érase', 'ñoño']
['árbol', 'cigüeña', 'érase', 'murciélago', 'ñoño', 'nunca', 'otoño']

```



# Referencias

- Wikipedia
- <http://www.inventwithpython.com/>
- <http://www.diveintopython3.net/>
- Documentación de Python:
- <https://docs.python.org/3/tutorial/index.html>
- <https://docs.python.org/3/>
- Marzal Varó, Andrés; Gracia Luengo, Isabel; García Sevilla, Pedro (2014). Introducción a la programación con Python 3. Disponible en: <http://dx.doi.org/10.6035/Sapientia93>
- <https://realpython.com/python-sets/>
- <https://realpython.com/python-dicts/>
- <https://realpython.com/python-lists-tuples/>