Tema 6:

Estructuras de datos básicas (6 h)

6. Estructuras de datos básicas (6h)

Profesor: Salvador Martínez Bolinches

Centro: IES Font de Sant Lluís

Año: 2025

Objetivos de aprendizaje

- Reconocer y utilizar los tipos de datos básicos en Python.
- Aplicar conversiones entre tipos.
- Manejar operadores aritméticos, relacionales, lógicos y de asignación.
- Identificar la importancia del tipado dinámico y fuerte. Resolver ejercicios prácticos usando operaciones básicas.

6. Estructuras de datos básicas.

En Python tenemos datos simples (**int**, **float**, **bool**, …) y datos secuencia. Los **tipos de datos secuencia** son los que se pueden recorrer, es decir, listas (**list**), tuplas (**tuple**), rangos (**range**), cadenas (**str**), **byte**, **bytearray**, **set**, y **frozenset**.

Las **cadenas de caracteres** también son un tipo de datos secuenciales e **inmutables**. Python no dispone del tipo de datos vector o **array**, de tamaño fijo y con elementos del mismo tipo. Por contra, si que dispone del tipo de datos **list**, de tamaño dinámico o variable y que puede contener elementos de diferentes tipos.

6.1. Listas

Una lista es una colección ordenada y **mutable** de elementos. Se definen con corchetes `[]` y pueden contener elementos heterogéneos.

```
numeros = [1, 2, 3, 4]
mixta = [1, "dos", 3.0, True]
```

Operaciones principales

1. Recorrer una lista

   for num in lista:

   print(num, end=””)

2. Recorrer dos listas

   for num, letra in zip(lista1, lista2): → *zip: une dos listas*

   print(num, “ ”, letra)

3. Operador pertenencia

   elem in lista → *True/False*

   elem not in lista → *True/false*

4. Concatenar listas o añadir elementos a una lista

   lista3 = lista1 + lista2

   lista1 = [1,2,3,4,5,6]
   lista1 + [7,8,9]

5. Repetir listas

   lista * num → repite la lista num veces

6. Indexar listas

   lista[pos] → Devuelve el valor del elemento situado en pos

   lista[-1] → Devuelve el valor del último elemento de la lista

7. Rebanar listas (slice)

   lista[pos1:pos2] → Devuelve los elementos de la lista desde pos1 hasta pos2-1

   lista[pos1:pos2:salto] → Elementos de la lista desde pos1 hasta pos2-1 de salto en salto

   lista[pos1:] → Devuelve los elementos de la lista desde pos1 hasta el final

   lista[:pos2] → Devuelve los elementos de la lista desde el principio hasta pos2-1

   lista[::-1] → Invierte la lista

8. Longitud de una lista

   len(lista)

9. Valor máximo de una lista

   max(lista)

10. Valor mínimo de una lista

    min(lista)

11. Suma de una lista

    sum(lista)

```python
# Indexación
print(numeros[0])     # 1
print(numeros[-1])    # 4

# Slicing
print(numeros[1:3])   # [2, 3]

# Métodos
numeros.append(5)     # [1,2,3,4,5]
numeros.remove(3)     # [1,2,4,5]
numeros.sort()        # ordena in place
```

12. Ordenar una lista

    sorted(lista) → En orden ascendente

    sorted(lista, reverse=True) → Ordena la lista en orden inverso

13. Tablas o listas de varias dimensiones

    tabla = [lista1,lista2,lista3]

    tabla =[[1,2,3],[4,5,6],[7,8,9]]

```
for fila in tabla:
    for elem in fila:
        print(elem)
```

14. Enumerar listas

    list(enumerate(lista)) → Devuelve una tupla (posición, elemento)

    enumerate(lista, start = 1) → La secuencia empieza en **start**

Y si son **mutables**, además puedo modificar un elemento o borrarlo.

Mutables	Inmutables
list, bytearray, set	tuple, range, str, byte, frozenset

Programación orientada a objetos (POO)

Los datos mutables usan la POO, es decir, son objetos que pueden usar métodos. Al ser mutables, me permiten hacer operaciones como:

lista[pos] = valor → modifica la lista cambiando un elemento

del lista[pos] → modifica la lista eliminando un elemento

Esto no se podía hacer con cadenas de caracteres, que son inmutables.

Si hago:

lista1 = lista2 → hay una sola lista **apuntada** por lista1 y lista2.

Si modifico una de las listas, la otra se modifica automáticamente, ya que es **la misma lista**.

Para poder tener dos listas independientes tengo que usar las rebanadas:

lista2 = lista1[:] → Crea lista2 igual a lista1 pero independiente

Métodos

lista. [Tab] → Lista de métodos para trabajar con listas.

```
salva@PC002:~$ python3
Python 3.10.12 (main, May 27 2025, 17:12:29) [GCC 11.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> lista=list()
>>> type(lista)
<class 'list'>
>>> lista.
lista.append(    lista.copy()     lista.extend(    lista.insert(    lista.remove(    lista.sort(
lista.clear()    lista.count(     lista.index(     lista.pop(       lista.reverse()
```

Métodos de inserción

- lista.append(elem) → Añade *elem* al final de la lista
- lista1.extend(lista2) → Concatena *lista1* y *lista2*
- lista.insert(pos,elem) → Añade *elem* a la *pos* de la lista

Métodos de borrado

- lista.pop() → Devuelve el último elemento y lo elimina de la lista
- lista.pop(pos) → Devuelve el elemento situado en *pos* y lo elimina de la lista
- lista.remove(elem) → Elimina de la lista la 1ª ocurrencia de *elem*

Métodos de ordenación

- lista.reverse() → Invierte la lista
- lista.sort() → Ordena la lista
- lista.sort(reverse=True) → Ordena la lista en orden inverso
- lista.sort(key=str.lower) → Criterio de ordenación

Métodos de búsqueda

- lista.count(elem) → Cuenta las ocurrencias de *elem* en la lista
- lista.index(elem) → Devuelve la *pos* de la 1ª ocurrencia de *elem* o *-1* si no está
- lista.index(elem,pos) → Busca *elem* a partir de *pos*

Métodos de copia

- lista2 = lista1.copy() → Copia lista1 a lista2 (independientes)

Operaciones avanzadas con secuencias:

- list(**map**(función, lista))→ Aplica la *función a* todos los elementos de la secuencia (*lista*)

  lista = [“1”,”2”,”3”]; list(map(int,lista)) → [1,2,3]

- list(**filter**(función, lista))→ Muestra los elementos de *lista* que cumplen la *función*

  def par(x): return x%2==0

  lista = [1,2,3,4,5,6]

  list(filter(par,lista)) → [2,4,6]

- **reduce**(función, lista) → Reduce la *lista* aplicando la *función*

  from functools import reduce

  def add(x,y): return x+y

  lista = [1,2,3,4,5]

  reduce(add, lista) → 15

- Listas comprimidas.

  [x ** 3 for x in [1,2,3,4,5]] → [1, 8, 27, 64, 125]

  [x for x in range(10) if x%2==0] → [0, 2, 4, 6, 8]

  [x + y for x in [1,2,3] for y in [4,5,6]] → [5, 6, 7, 6, 7, 8, 7, 8, 9]

Función map

`map(funcion,secuencia)`: Ejecuta la función enviada por parámetro sobre cada uno de los elementos de la secuencia.

Ejemplo

```
>>> items = [1, 2, 3, 4, 5]
>>> def sqr(x): return x ** 2
>>> list(map(sqr, items))
[1, 4, 9, 16, 25]
```

Función filter

`filter(funcion,secuencia)`: Devuelve una secuencia con los elementos de la secuencia envíada por parámetro que devuelvan `True` al aplicarle la función envíada también como parámetro.

Ejemplo

```
>>> lista = [1,2,3,4,5]
>>> def par(x): return x % 2==0
>>> list(filter(par,lista))
```

Función reduce

`reduce(funcion,secuencia)`: Devuelve un único valor que es el resultado de aplicar la función á los lementos de la secuencia.

Ejemplo

```
>>> from functools import reduce
>>> lista = [1,2,3,4,5]
>>> def add(x,y): return x + y
>>> reduce(add,lista)
15
```

list comprehension

`list comprehension` nos propociona una alternativa para la creación de listas. Es parecida a la función `map`, pero mientras `map` ejecuta una función por cada elemento de la secuencia, con esta técnica se aplica una expresión.

Ejemplo

```
>>> [x ** 3 for x in [1,2,3,4,5]]
[1, 8, 27, 64, 125]

>>> [x for x in range(10) if x % 2 == 0]
[0, 2, 4, 6, 8]

>>> [x + y for x in [1,2,3] for y in [4,5,6]]
[5, 6, 7, 6, 7, 8, 7, 8, 9]
```

6.2. Tuplas

Una tupla es una colección **ordenada** e **inmutable** de elementos. Se definen con paréntesis **()** y pueden contener elementos heterogéneos. Se usan menos que las listas y son útiles para datos que no deben cambiar. Una vez creada, no se puede modificar (modificar un elemento o eliminar un elemento).

```
coordenadas = (10, 20)
print(coordenadas[0])  # 10
```

Operaciones principales

Con las tuplas podemos realizar las mismas operaciones que con las listas, excepto las de inserción o borrado. Hay comandos de Python que devuelven tuplas. Por ejemplo:

```
>>> cociente,resto = divmod(7,2) → cociente=3    resto=1
```

Construcción de una tupla

Para crear una lista puedo usar varias formas:

Con los caracteres (y):

```
>>> tupla1 = ()
>>> tupla2 = ("a",1,True)
```

Utilizando el constructor tuple, que toma como parámetro un dato de algún tipo secuencia.

```
>>> tupla3=tuple()
>>> tuple4=tuple([1,2,3])
```

Empaquetado y desempaquetado de tuplas

Si a una variable se le asigna una secuencia de valores separados por comas, el valor de esa variable será la tupla formada por todos los valores asignados.

- `>>> tuple = 1,2,3`
- `>>> tuple`
- `(1, 2, 3)`
- `>>> a,b,c=tuple`
- `>>> a`
- `1`

Si se tiene una tupla de longitud k, se puede asignar la tupla a k variables distintas y en cada variable quedará una de las componentes de la tupla.

Operaciones básicas con tuplas

En las tuplas se pueden realizar las siguientes operaciones:

- Las tuplas se pueden recorrer.
- Operadores de pertenencia: `in` y `not in`.
- Concatenación: +
- Repetición: *
- Indexación
- Slice

Entre las funciones definidas podemos usar: `len`, `max`, `min`, `sum`, `sorted`.

Las tuplas son inmutables

```
>>> tupla = (1,2,3)
>>> tupla[1]=5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Métodos principales

Métodos de búsqueda: `count`, `index`

```
>>> tupla = (1,2,3,4,1,2,3)
>>> tupla.count(1)
2
>>> tupla.index(2)
1
>>> tupla.index(2,2)
5
```

6.3. Rangos

Definición de un rango. Constructor range

Al crear un rango (secuencia de números) obtenemos un objeto que es de la clase `range`:

```
>>> rango = range(0,10,2)
>>> type(rango)
<class 'range'>
```

Veamos algunos ejemplos, convirtirndo el rango en lista para ver la secuencia:

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(1, 11))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> list(range(0, 30, 5))
[0, 5, 10, 15, 20, 25]
>>> list(range(0, 10, 3))
[0, 3, 6, 9]
>>> list(range(0, -10, -1))
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
```

Recorrido de un rango

Los rangos se suelen usar para ser recorrido, cuando tengo que crear un bucle cuyo número de iteraciones lo se de antemanos puedo usar una estructura como esta:

```
>>> for i in range(11):
...     print(i,end=" ")
0 1 2 3 4 5 6 7 8 9 10
```

Operaciones básicas con range

En las tuplas se pueden realizar las siguientes operaciones:

- Los rangos se pueden recorrer.
- Operadores de pertenencia: `in` y `not in`.
- Indexación
- Slice

Entre las funciones definidas podemos usar: `len`, `max`, `min`, `sum`, `sorted`.

Ademas un objeto `range` posee tres atributos que nos almacenan el comienzo, final e intervalo del rango:

```
>>> rango = range(1,11,2)
>>> rango.start
1
>>> rango.stop
11
>>> rango.step
2
```

6.4. Cadenas de caracteres

Las cadenas de caracteres no son realmente un tipo básico, sino que también son un tipo de dato secuencia.

Definición de cadenas. Constructor str

Podemos definir una cadena de caracteres de distintas formas:

```
>>> cad1 = "Hola"
>>> cad2 = '¿Qué tal?'
>>> cad3 = '''Hola,
    que tal?'''
```

También podemos crear cadenas con el constructor `str` a partir de otros tipos de datos.

```
>>> cad1=str(1)
>>> cad2=str(2.45)
>>> cad3=str([1,2,3])
```

Operaciones básicas con cadenas de caracteres

Como veíamos en el apartado “Tipo de datos secuencia” podemos realizar las siguientes operaciones:

- Las cadenas se pueden recorrer.
- Operadores de pertenencia: `in` y `not in`.
- Concatenación: +
- Repetición: *
- Indexación
- Slice

Entre las funciones definidas podemos usar: `len`, `max`, `min`, `sorted`.

Las cadenas son inmutables

```
>>> cad = "Hola que tal?"
>>> cad[4]="."
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Comparación de cadenas

Las cadenas se comparan carácter a carácter, en el momento en que dos caracteres no son iguales se compara alfabéticamente (es decir, se convierte a código unicode y se comparan).

Ejemplos:

```
>>> "a">"A"
True
>>> ord("a")
97
>>> ord("A")
65
>>> "informatica">"informacion"
True
>>> "abcde">"abcdef"
False
```

Funciones repr, ascii, bin

- `repr(objeto)`: Devuelve una cadena de caracteres que representa la información de un objeto.

```
>>> repr(range(10))
'range(0, 10)'
>>> repr("piña")
"'piña'"
```

  La cadena devuelta por `repr()` debería ser aquella que, pasada a `eval()`, devuelve el mismo objeto.

```
>>> type(eval(repr(range(10))))
<class 'range'>
```

- `ascii(objeto)`: Devuelve también la representación en cadena de un objeto pero en este caso muestra los caracteres con un código de escape . Por ejemplo en ascii (Latin1) la `á` se presenta con `\xe1`.

```
>>> ascii("á")
"'\\xe1'"
>>> ascii("piña")
"'pi\\xf1a'"
```

- `bin(numero)`: Devuelve una cadena de caracteres que corresponde a la representación binaria del número recibido.

```
>>> bin(213)
'0b11010101'
```

Métodos principales de cadenas

Métodos de formato:

```
>>> cad = "hola, como estás?"
>>> print(cad.capitalize())
Hola, como estás?
>>> cad = "Hola Mundo"
>>> print(cad.lower())
hola mundo
>>> cad = "hola mundo"
>>> print(cad.upper())
HOLA MUNDO
>>> cad = "Hola Mundo"
>>> print(cad.swapcase())
hOLA mUNDO
>>> cad = "hola mundo"
>>> print(cad.title())
Hola Mundo
>>> print(cad.center(50))
                    hola mundo
>>> print(cad.center(50,"="))
====================hola mundo====================
>>> print(cad.ljust(50,"="))
hola mundo========================================
>>> print(cad.rjust(50,"="))
========================================hola mundo
>>> num = 123
>>> print(str(num).zfill(12))
000000000123
```

```
cadena.capitalize    cadena.isalnum
cadena.casefold      cadena.isalpha
cadena.center        cadena.isdecimal
cadena.count         cadena.isdigit
cadena.encode        cadena.isidentifier
cadena.endswith      cadena.islower
cadena.expandtabs    cadena.isnumeric
cadena.find          cadena.isprintable
cadena.format        cadena.isspace
cadena.format_map    cadena.istitle
cadena.index         cadena.isupper

cadena.join          cadena.rsplit
cadena.ljust         cadena.rstrip
cadena.lower         cadena.split
cadena.lstrip        cadena.splitlines
cadena.maketrans     cadena.startswith
cadena.partition     cadena.strip
cadena.replace       cadena.swapcase
cadena.rfind         cadena.title
cadena.rindex        cadena.translate
cadena.rjust         cadena.upper
cadena.rpartition    cadena.zfill
```

Métodos de búsqueda:

```
>>> cad = "bienvenido a mi aplicación"
>>> cad.count("a")
3
>>> cad.count("a",16)
2
>>> cad.count("a",10,16)
1
>>> cad.find("mi")
13
>>> cad.find("hola")
-1
>>> cad.rfind("a")
21
```

El método `index()` y `rindex()` son similares a los anteriores pero provocan una excepción `ValueError` cuando no encuentra la subcadena.

Métodos de validación:

- cadena.startswith(“subcadena”) → True/False
- cadena.startswith(“subcadena”, pos) → True/False A partir de pos
- cadena.endswith(“subcadena”) → True/False
- cadena.endswith(“subcadena”, pos1, pos2) → True/False
- isdigit(), islower(), isupper(), isspace(), istitle(), … → True/False

Métodos de substitución:

- replace(cad_original, cad_nueva)
- cadena.strip() → Elimina espacios delante y detrás
- cadena.strip(“char”) → Elimina char delante y detrás

Métodos de unión y división:

- cadena.split(“:”) → Devuelve una lista de las subcadenas separadas por :
- cadena.splitlines() → Devuelve una lista de las líneas de cadena

La codificación de caracteres en python3

En Python 3.x las cadenas de caracteres pueden ser de tres tipos: Unicode, Byte y Bytearray.

- El tipo `unicode` permite caracteres de múltiples lenguajes y cada carácter en una cadena tendrá un valor inmutable.
- El tipo `byte` sólo permitirá caracteres ASCII y los caracteres son también inmutables.
- El tipo `bytearray` es como el tipo `byte` pero, en este caso, los caracteres de una cadena si son mutables.

Algo que debe entenderse (e insiste Mark Pilgrim en su libro *Dive into Python*) es que “los bytes no son caracteres, los bytes son bytes; un carácter es en realidad una abstracción; y una cadena de caracteres es una sucesión de abstracciones”.

Funciones chr() y ord()

- `chr(i)`: Nos devuelve el carácter Unicode que representa el código `i`.

```
>>> chr(97)
'a'
>>> chr(1004)
'6'
```

- `ord(c)`: recibe un carácter `c` y devuelve el código unicode correspondiente.

```
>>> ord("a")
97
>>> ord("6")
1004
```

6.5. Tipos de datos binarios (*bytes y bytearray*)

Bytes

El tipo `bytes` es una secuencia inmutable de bytes. Solo admiten caracteres ASCII. También se pueden representar los bytes con números enteros cuyo valores deben cumplir `0 <= x < 256`.

Definición de bytes. Constructor bytes

Podemos definir un tipo `bytes` de distintas formas:

```
>>> byte1 = b"Hola"
>>> byte2 = b'¿Qué tal?'
>>> byte3 = b'''Hola,
    que tal?'''
```

También podemos crear cadenas con el constructor bytes a partir de otros tipos de datos.

```
>>> byte1=bytes(10)
>>> byte1
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
>>> byte2=bytes(range(10))
>>> byte2
b'\x00\x01\x02\x03\x04\x05\x06\x07\x08\t'
>>> byte3=bytes.fromhex('2Ef0 F1f2')
>>> byte3
b'.\xf0\xf1\xf2'
```

Bytearray

El tipo `bytearray` es un tipo mutable de bytes.

Definición de bytearray. Constructor bytearray

```
>>> ba1=bytearray()
>>> ba1
bytearray(b'')
>>> ba2=bytearray(10)
>>> ba2
bytearray(b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00')
>>> ba3=bytearray(range(10))
>>> ba3
bytearray(b'\x00\x01\x02\x03\x04\x05\x06\x07\x08\t')
>>> ba4=bytearray(b"hola")
>>> ba4
bytearray(b'hola')
>>> ba5=bytearray.fromhex('2Ef0 F1f2')
>>> ba5
bytearray(b'.\xf0\xf1\xf2')
```

Operaciones básicas con bytes y bytearray

Como veíamos en el apartado “Tipo de datos secuencia” podemos realizar las siguientes operaciones:

- Recorrido
- Operadores de pertenencia: `in` y `not in`.
- Concatenación: +
- Repetición: *
- Indexación
- Slice

Entre las funciones definidas podemos usar: `len`, `max`, `min`, `sum`, `sorted`.

Los bytes son inmutables, los bytearray son mutables

```
>>> byte=b"hola"
>>> byte[2]=b'g'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'bytes' object does not support item assignment
>>> ba1=bytearray(b'hola')
>>> ba1[2]=123
>>> ba1
bytearray(b'ho{a')
>>> del ba1[3]
>>> ba1
bytearray(b'ho{')
```

Métodos de bytes y bytearray

```
byte1.capitalize  byte1.index       byte1.join        byte1.rindex
byte1.strip
byte1.center      byte1.isalnum     byte1.ljust       byte1.rjust
byte1.swapcase
byte1.count       byte1.isalpha     byte1.lower       byte1.rpartition
byte1.title
byte1.decode      byte1.isdigit     byte1.lstrip      byte1.rsplit
byte1.translate
byte1.endswith    byte1.islower     byte1.maketrans   byte1.rstrip
byte1.upper
byte1.expandtabs  byte1.isspace     byte1.partition   byte1.split
byte1.zfill
byte1.find        byte1.istitle     byte1.replace     byte1.splitlines
byte1.fromhex     byte1.isupper     byte1.rfind       byte1.startswith
```

```
bytearray1.append      bytearray1.index       bytearray1.lstrip
bytearray1.rstrip
bytearray1.capitalize  bytearray1.insert      bytearray1.maketrans
bytearray1.split
bytearray1.center      bytearray1.isalnum     bytearray1.partition
bytearray1.splitlines
bytearray1.clear       bytearray1.isalpha     bytearray1.pop
bytearray1.startswith
bytearray1.copy        bytearray1.isdigit     bytearray1.remove
bytearray1.strip
bytearray1.count       bytearray1.islower     bytearray1.replace
bytearray1.swapcase
bytearray1.decode      bytearray1.isspace     bytearray1.reverse
bytearray1.title
bytearray1.endswith    bytearray1.istitle     bytearray1.rfind
bytearray1.translate
bytearray1.expandtabs  bytearray1.isupper     bytearray1.rindex
bytearray1.upper
bytearray1.extend      bytearray1.join        bytearray1.rjust
bytearray1.zfill
bytearray1.find        bytearray1.ljust       bytearray1.rpartition
bytearray1.fromhex     bytearray1.lower       bytearray1.rsplit
```

Si nos fijamos la mayoría de los métodos en el caso de los `bytes` son los de las cadenas de caracteres, y en los `bytearray` encontramos también métodos propios de las listas.

Métodos encode y decode

Los caracteres cuyo código es mayor que 256 no se pueden usar para representar los bytes, sin embargo si podemos indicar una codificación de caracteres determinada para que ese carácter se convierte en un conjunto de bytes.

```
>>> byte1=b'piña'
  File "<stdin>", line 1
SyntaxError: bytes can only contain ASCII literal characters.
>>> byte1=bytes('piña',"utf-8")
>>> byte1
b'pi\xc3\xb1a'
>>> len(byte1)
5
>>> byte1=bytes('piña',"latin1")
>>> byte1
b'pi\xf1a'
```

Podemos también convertir una cadena unicode a bytes utilizando el método encode:

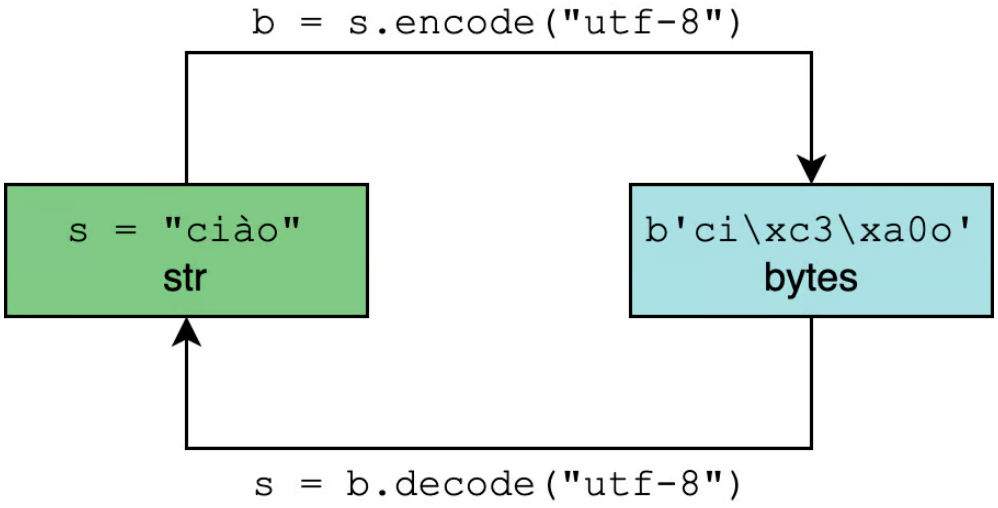
```
>>> cad="piña"
>>> byte1=cad.encode("utf-8")
>>> byte1
b'pi\xc3\xb1a'
```

Para hacer la función inversa, convertir de bytes a unicode utilizamos el método decode:

```
>>> byte1.decode("utf-8")
'piña'
```

El problema lo tenemos si hemos codificado utilizando un código e intentamos decodificar usando otro.

```
>>> byte1=bytes('piña',"latin1")
>>> byte1.decode("utf-8")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xf1 in position 2: invalid
continuation byte
>>> byte1.decode("utf-8","ignore")
'pia'
>>> byte1.decode("utf-8","replace")
'piña'
```



6.6. Tipos de datos conjuntos (*set y frozenset*)

Set

Los conjuntos (`set`): Me permiten guardar conjuntos (desornedos) de datos (a los que se puede calcular una función hash), en los que no existen repeticiones. Es un tipo de datos mutable.

Normalmente se usan para comprobar si existe un elemento en el conjunto, eliminar duplicados y cálculos matématicos, como la intersección, unión, diferencia,…

Definición de set. Constructor set

Podemos definir un tipo `set` de distintas formas:

```
>>> set1 = set()
>>> set1
set()
>>> set2=set([1,1,2,2,3,3])
>>> set2
{1, 2, 3}
>>> set3={1,2,3}
>>> set3
{1, 2, 3}
```

Frozenset

El tipo `frozenset` es un tipo inmutable de conjuntos.

Definición de frozenset. Constructor frozenset

```
>>> fs1=frozenset()
>>> fs1
frozenset()
>>> fs2=frozenset([1,1,2,2,3,3])
>>> fs2
frozenset({1, 2, 3})
```

Operaciones básicas con set y frozenset

De las operaciones que estudiamos en el apartado “Tipo de datos secuencia” los conjuntos sólo aceptan las siguientes:

- Recorrido
- Operadores de pertenencia: `in` y `not in`.

Entre las funciones definidas podemos usar: `len`, `max`, `min`, `sorted`.

Los set son mutables, los frozenset son inmutables

```
>>> set1={1,2,3}
>>> set1.add(4)
>>> set1
{1, 2, 3, 4}
>>> set1.remove(2)
>>> set1
{1, 3, 4}
```

El tipo `frozenset` es inmutable por lo tanto no posee los métodos `add` y `remove`.

Métodos de set y frozenset

```
set1.add                    set1.issubset                    set1.isdisjoint
set1.clear                  set1.issuperset                  set1.intersection_update
set1.copy                   set1.pop                         set1.update
set1.difference             set1.remove                      set1.intersection
set1.difference_update      set1.symmetric_difference        set1.union
set1.discard                set1.symmetric_difference_update
```

Veamos algunos métodos, partiendo siempre de estos dos conjuntos:

```
>>> set1={1,2,3}
>>> set2={2,3,4}
>>> set1.difference(set2)
{1}
>>> set1.difference_update(set2)
>>> set1
{1}
>>> set1.symmetric_difference(set2)
{1, 4}
>>> set1.symmetric_difference_update(set2)
>>> set1
{1, 4}
>>> set1.intersection(set2)
{2, 3}
>>> set1.intersection_update(set2)
>>> set1
{2, 3}
>>> set1.union(set2)
{1, 2, 3, 4}
>>> set1.update(set2)
>>> set1
{1, 2, 3, 4}
```

Veamos los métodos de añadir y eliminar elementos:

```
>>> set1 = set()
>>> set1.add(1)
>>> set1.add(2)
>>> set1
{1, 2}
>>> set1.discard(3)
>>> set1.remove(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 3
>>> set1.pop()
1
>>> set1
{2}
```

Y los métodos de comprobación:

```
>>> set1 = {1,2,3}
>>> set2 = {1,2,3,4}
>>> set1.isdisjoint(set2)
False
>>> set1.issubset(set2)
True
>>> set1.issuperset(set2)
False
>>> set2.issuperset(set1)
True
```

Por último los métodos de `frozenset`:

```
fset1.copy                  fset1.isdisjoint            fset1.symmetric_difference
fset1.difference            fset1.issubset              fset1.union
fset1.intersection          fset1.issuperset
```

6.7. Tipo de datos iterador y generador

Iteradores

Un objeto iterable es aquel que puede devolver un iterador. Normalmente las colecciones que hemos estudiados son iterables. Un iterador me permite recorrer los elementos del objeto iterable.

Definición de iterador. Constructor iter

```
>>> iter1 = iter([1,2,3])
>>> type(iter1)
<class 'list_iterator'>
>>> iter2 = iter("hola")
>>> type(iter2)
<class 'str_iterator'>
```

Función next(), reversed()

Para recorrer el iterador, utilizamos la función `next()`:

```
>>> next(iter1)
1
>>> next(iter1)
2
>>> next(iter1)
3
>>> next(iter1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

La función `reversed()` devuelve un iterador con los elementos invertidos, desde el último al primero.

```
>>> iter2 = reversed([1,2,3])
>>> next(iter2)
3
>>> next(iter2)
2
>>> next(iter2)
1
>>> next(iter2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

El módulo itertools

El módulo itertools contiene distintas funciones que nos devuelven iteradores.

Veamos algunos ejemplos:

`count()`: Devuelve un iterador infinito.

```
>>> from itertools import count
>>> counter = count(start=13)
>>> next(counter)
13
>>> next(counter)
14
```

`cycle()`: devuelve una secuencia infinita.

```
>>> from itertools import cycle
>>> colors = cycle(['red', 'white', 'blue'])
>>> next(colors)
'red'
>>> next(colors)
'white'
>>> next(colors)
'blue'
>>> next(colors)
'red'
```

`islice()`: Retorna un iterador finito.

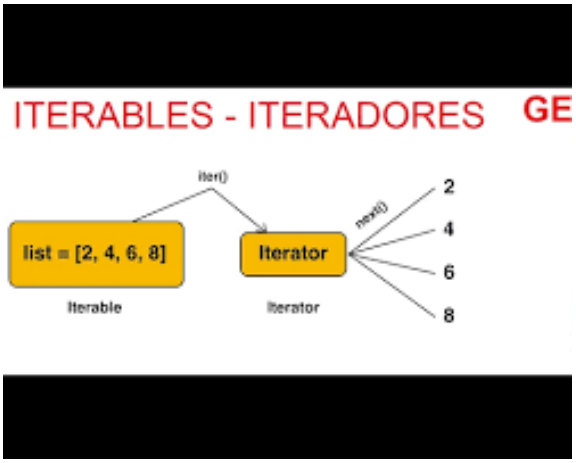
```
>>> from itertools import islice
>>> limited = islice(colors, 0, 4)
>>> for x in limited:
...   print(x)
white
blue
red
white
```

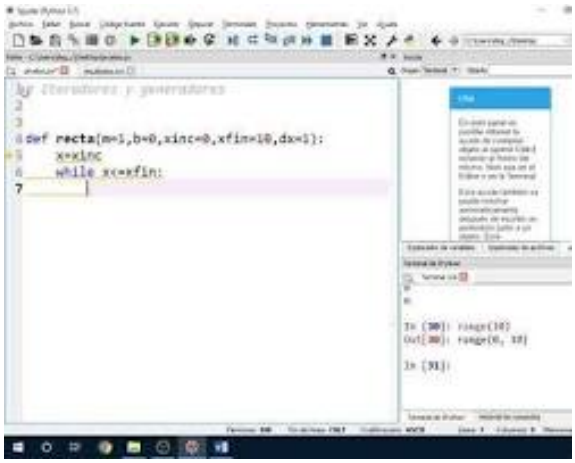
Generadores

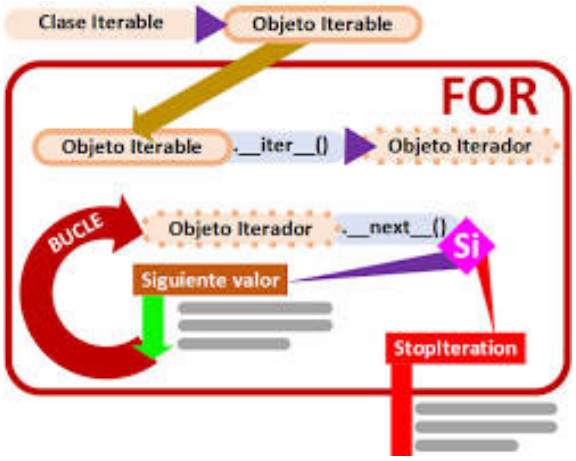
Un generador es un tipo concreto de iterador. Es una función que permite obtener sus resultados paso a paso. Por ejemplo, hacer una función que cada vez que la llamemos nos de el próximo número par. Tenemos dos maneras de crear generadores:

1. Realizar una función que devuelva los valores con la palabra reservada `yield`. Lo veremos con profundidad cuando estudiemos las funciones.
2. Utilizando la sintaxis de las “list comprehension”. Por ejemplo:

```
>>> iter1 = (x for x in range(10) if x % 2==0)
>>> next(iter1)
0
>>> next(iter1)
2
>>> next(iter1)
4
```





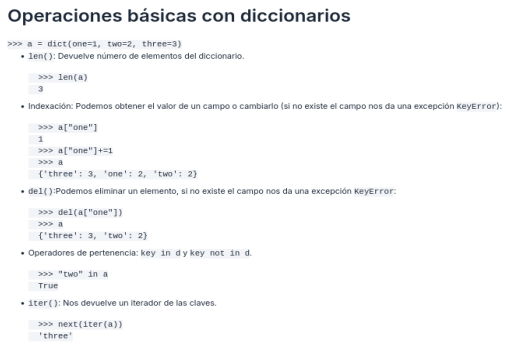


6.8. Diccionarios

Es un **tipo de datos mapas**. Los diccionarios son una colección **desordenada** de **pares clave:valor**.

- Se definen con `{clave: valor}`.
- El tipo de datos es **dict**.
- Muy útiles para representar entidades u objetos simples.

```
persona = {"nombre": "Ana", "edad": 25}
print(persona["nombre"])  # Ana
```



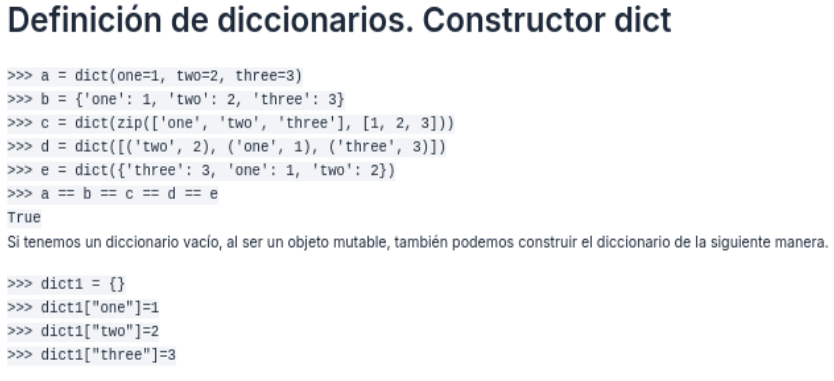
Operaciones principales

```
persona["edad"] = 26        # modificar
persona["ciudad"] = "Madrid"  # añadir
del persona["ciudad"]       # eliminar
print(persona.keys())       # dict_keys(['nombre','edad'])
print(persona.values())     # dict_values(['Ana', 26])
```

- dic = {} → Crea un diccionario vacío
- len(dic) → Tamaño del diccionario
- del dic(“clave”) → Borra un elemento del diccionario
- clave in dic → True/False si existe la clave
- dic2=dic1 → El mismo diccionario apuntado por dos nombres

Métodos principales de diccionarios

- dic.clear() → Borra todos los datos del diccionarios
- dic2=dic1.copy() → Copia dic1 a dic2. Dos diccionarios independientes
- dic1.update(dic2) → Actualiza dic1 con dic2 (los añade)
- dic.get(clave) → Devuelve el valor asociado a clave o **nada si no existe**.
- dic.get(clave, “No”) → Devuelve el valor asociado a clave o “**No**” **si no existe**.
- dic[clave] → Devuelve el valor asociado a clave o la **excepción KeyError si no existe**.
- dic.pop(clave) → Devuelve el valor de clave y lo borra de la lista o KeyError si no existe
- dic.pop(clave, “No”) → Devuelve el valor de clave y lo borra de la lista o “No” si no existe
- for clave in dic.keys(): → Recorrer las claves
- for valor in dic.values(): → Recorrer los valores
- for cla,val in dic.items(): → Recorrer las claves y valores



Métodos de eliminación: clear

```
>>> dict1 = dict(one=1, two=2, three=3)
>>> dict1.clear()
>>> dict1
{}
```

Métodos de agregado y creación: copy, dict.fromkeys, update, setdefault

```
>>> dict1 = dict(one=1, two=2, three=3)
>>> dict2 = dict1.copy()
>>> dict.fromkeys(["one","two","three"])
{'one': None, 'two': None, 'three': None}
>>> dict.fromkeys(["one","two","three"],100)
{'one': 100, 'two': 100, 'three': 100}
>>> dict1 = dict(one=1, two=2, three=3)
>>> dict2 = {'four':4,'five':5}
>>> dict1.update(dict2)
>>> dict1
{'one': 1, 'two': 2, 'three': 3, 'four': 4, 'five': 5}
>>> dict1 = dict(one=1, two=2, three=3)
>>> dict1.setdefault("four",4)
4
>>> dict1
{'one': 1, 'two': 2, 'three': 3, 'four': 4}
>>> dict1.setdefault("one",-1)
1
>>> dict1
{'one': 1, 'two': 2, 'three': 3, 'four': 4}
```

Métodos de retorno: get, pop, popitem, items, keys, values

```
>>> dict1 = dict(one=1, two=2, three=3)
>>> dict1.get("one")
1
>>> dict1.get("four")
>>> dict1.get("four","no existe")
'no existe'
>>> dict1.pop("one")
1
>>> dict1
{'two': 2, 'three': 3}
>>> dict1.pop("four")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'four'
>>> dict1.pop("four","no existe")
'no existe'
>>> dict1 = dict(one=1, two=2, three=3)
>>> dict1.popitem()
('one', 1)
>>> dict1
{'two': 2, 'three': 3}
>>> dict1 = dict(one=1, two=2, three=3)
>>> dict1.items()
dict_items([('one', 1), ('two', 2), ('three', 3)])
>>> dict1.keys()
dict_keys(['one', 'two', 'three'])
```

El tipo de datos dictviews

Los tres últimos métodos devuelven un objeto de tipo `dictviews`.

Esto devuelve una vista dinámica del ciccionario, por ejemplo:

```
>>> dict1 = dict(one=1, two=2, three=3)
>>> i = dict1.items()
>>> i
dict_items([('one', 1), ('two', 2), ('three', 3)])
>>> dict1["four"]=4
>>> i
dict_items([('one', 1), ('two', 2), ('three', 3), ('four', 4)])
```

Es este tipo de datos podemos usar las siguientes funciones:

- `len()`: Devuelve número de elementos de la vista.
- `iter()`: Nos devuelve un iterador de las claves, valores o ambas.
- `x in dictview`: Devuelve True si x está en las claves o valores.

Recorrido de diccionarios

Podemos recorrer las claves:

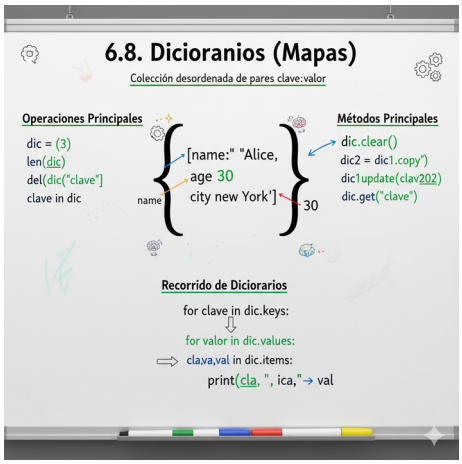
```
>>> for clave in dict1.keys():
...     print(clave)
one
two
three
```

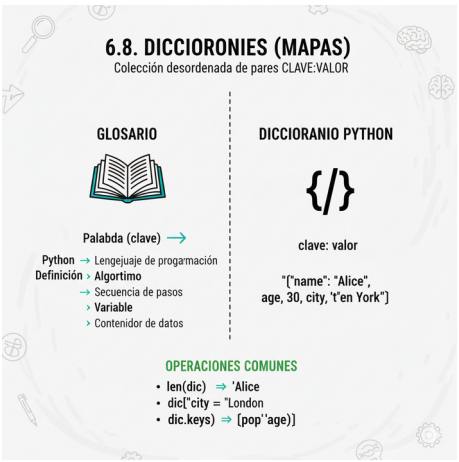
Podemos recorrer los valores:

```
>>> for valor in dict1.values():
...     print(valor)
1
2
3
```

O podemos recorrer ambos:

```
>>> for clave,valor in dict1.items():
...     print(clave,"->",valor)
one -> 1
two -> 2
three -> 3
```





6.9. Iteración básica sobre colecciones

Listas

```
for numero in [1, 2, 3]:
    print(numero)
```

Diccionarios

```
for clave, valor in persona.items():
    print(clave, "=>", valor)
```

Aunque los bucles se tratan en detalle en otro módulo, aquí se introduce la **iteración simple**.

6.10. Parte práctica (ejemplos y ejercicios)

Ejemplo 1: Manipulación de listas

```
nombres = ["Ana", "Luis", "Marta"]
nombres.append("Pedro")
print(nombres[1:3])  # ['Luis','Marta']
```

Ejemplo 2: Uso de tuplas

```
punto = (4, 5)
x, y = punto
print("x:", x, "y:", y)
```

Ejemplo 3: Operaciones con conjuntos

```
asignaturas = {"Matemáticas", "Lengua", "Historia"}
asignaturas.add("Inglés")
print("Inglés" in asignaturas)  # True
```

Ejemplo 4: Diccionarios

```
alumno = {"nombre": "Carlos", "notas": [7, 8, 9]}
print(alumno["notas"][1])  # 8
```

Ejercicios propuestos

1. Crea una lista con 5 productos. Añade un producto nuevo, elimina otro y muestra el resultado final.
2. Crea una tupla con 3 coordenadas (x, y, z). Desempaqueta la tupla en variables y muéstralas.
3. Crea dos conjuntos: alumnos de *Matemáticas* y de *Inglés*. Muestra los alumnos que estudian ambas asignaturas (intersección).
4. Crea un diccionario con nombres de alumnos y nota. Añade un alumno nuevo y modifica la nota de otro. Itera sobre el diccionario mostrando: `"Alumno: X - Nota: Y"`.
5. Crea un diccionario llamado `agenda` donde cada clave sea el nombre de un contacto y el valor un número de teléfono. Permite al usuario: añadir un contacto, buscar un número por nombre y eliminar un contacto.
6. ¿Cuál es el resultado de `3 + 4 * 2` y por qué?
7. Explica el resultado de:

   x = 5
   print(1 < x < 10)
8. Pide un número al usuario y muestra si es par o impar usando el operador %.
9. ¿Qué devuelve cada operación?
   - `10 // 3`
   - `10 % 3`
   - `2 ** 4`
10. Evalúa estas expresiones lógicas:
    - `5 > 2`
    - `7 == 7`
    - `4 != 4`

6.11. Distribución sugerida del tiempo (6 h)

- **Teoría (3 h)** → listas, tuplas, conjuntos, diccionarios.
- **Ejemplos guiados (1 h)** → demostraciones prácticas en el intérprete.
- **Ejercicios prácticos (2 h)** → resolución individual y en grupos, con puesta en común.