CRIPTOGRAFÍA CON PYTHON

Python avanzado

Autor: Enrique Melchor Iborra Sanjaime (em.iborrasanjaime@edu.gva.es)

Contenido

1. Estructuras avanzadas en Python

En Python, los datos se dividen en **mutables** e **inmutables**, dependiendo de si su contenido puede cambiar después de su creación

Los objetos **inmutables** no pueden cambiar su contenido después de ser creados. Cualquier "modificación" realmente crea un nuevo objeto en memoria.

Ejemplos de datos inmutables:

- Números (`int`, `float`, `complex`)
- Cadenas de texto (`str`)
- Tuplas (`tuple`)
- Conjuntos inmutables (`frozenset`)
- Booleanos (`bool`)

Los objetos **mutables** pueden cambiar su contenido sin cambiar su identidad en memoria.

Ejemplos de datos mutables:

- Listas (`list`)
- Diccionarios (`dict`)
- Conjuntos (`set`)
- Objetos definidos por el usuario (clases)

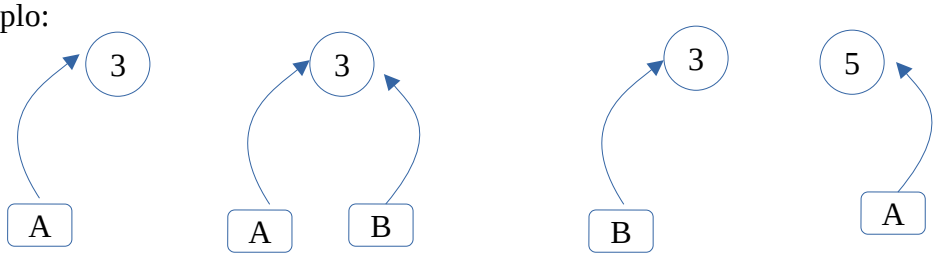
Objetos inmutables (int, strings, boolean, …..)

`a=3` # Se crea el literal 3 en memoria y se asigna su “direccion” a la variable a

`b=3` # como el literal 3 ya existe, se le asigna su “direccion” a la varible b

`a=5` # Se crea el literal 5 en memoria y se asigna su “direccion” a la variable a

Ejemplo:



Por “direccion” , se refiere a un identificador de objeto (no una dirección de memoria.

Este identificador se puede obtener con la funcio `id(var)`

Prueba:

```
>>>|a=9
>>>|b=10
>>>|id(a)
   |11754152
>>>|id(b)
   |11754184
>>>|b=a
>>>|id(b)
   |11754152
```

En Pyton se produce **aliasing** , es decir, los objetos tienen individualidad, y múltiples nombres

Objetos mutables (listas, diccionarios, conjuntos, user defined...)

En Python, las variables son referencias a objetos en memoria, no contenedores de datos directamente. Cuando haces una asignación, como:

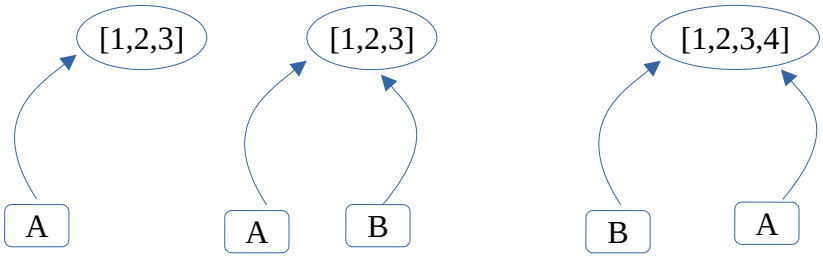
```
a = [1, 2, 3]       # a es una lista,  por lo tanto mutable
b = a
```

Tanto `a` como `b` apuntan al mismo objeto (una lista en este caso). No se crea una copia de la lista; simplemente, ambas variables hacen referencia al mismo objeto. Si modificas la lista a través de `b`:

```
b.append(4)   # esto añade un elemento al final de la lista
```

La lista a la que `a` apunta también se ve modificada, ya que `a` y `b` hacen referencia al mismo objeto en memoria.

Referencias y aliasing

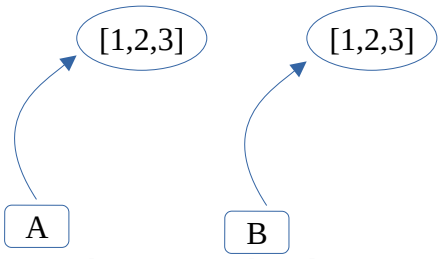


La diferencia con los objetos inmutables, es que cuando modificamos un objeto mutable, no se crea uno nuevo (si no existe) como pasa con los inmutables. Se modifica el objeto y se mantienen las referencias.

Orra diferencia con los inmutables, es que cuando asignamos una lista a una variable, SE CREA una nueva lista (aunque ya exista alguna lista con los mismos elementos).

a = [1, 2, 3]

b = [1, 2, 3]



Esto no puede pasar con los objetos inmutables. En el caso de dos objetos inmutables iguales, se apunta al mismo.

1.1. Listas

$$objetos = [\ 7, 'Hola', True, 3.5\ ]$$

Las **listas** son estructuras de datos que pueden almacenar varios elementos.

7	'Hola'	True	3.5
0	1	2	3

- Son ordenadas. Soportan indexación y slicing
- Son **Mutables**: Los elementos se pueden modificar después de la creación. (dinámicas)
- Se pueden agregar o eliminar elementos con métodos como append(), remove(), pop(), etc
- Los elementos de las listas pueden ser mutables o inmutables
- Se pueden anidar

```
frutas = ["manzana", "banana", "cereza"]    #Crea una lista
lista = list("1234")                        #Crea una lista
lista = [1, "Hola", 3.67, [1, 2, 3]]      # puede contener tipos distintos
print(frutas[0])         # Imprime 'manzana', primer elemento de la lista
frutas[0]="pera"                    # sustituye el elemento 0 (primer elemento)
del frutas[1]                       # borra el segundo elemento
frutas.append("naranja")            # Añade un elemento al final
print(frutas)   print(*frutas)      print(*frutas, sep=' -> ')
len(frutas)                         # devuelve numero de elementos de la lista
max(frutas) min(frutas)             # si los tipos son compatibles
sum(milista)                        # si todos son numeros
milista=[]                          # asignacion inicial vacia
milista[0]=24 , milista[1]=46       # milista debe tener al menos 2 elementos!!
milista[-1]=33 , milista[-2]=57     # -1 es el ultimo elemento de la lista
nuevafruta = frutas[2:4]            # 2:4 es un slice , va desde 2 a 4-1
nuevafruta2 = frutas[ :4],  nuevafruta3 = frutas[ 2:]
milista2 = [1,4,7,"platano"]        # elementos de tipos diferentes
mi_lista.insert(1, 15)              # Inserta 15 en el índice 1
mi_lista.extend([7, 8, 9])  # Agrega todos los elementos de otra lista al final
de la lista actual.
mi_lista.remove(15)                 # Elimina el valor 15
mi_lista.clear()                    # vacia la lista
ultimo = mi_lista.pop()    # devuelve el ultimo elemento y lo quita de la lista
otro = mi_lista.pop(2)     # devuelve el tercer elemento y lo quita de la lista
print(10 in mi_lista)      # Resultado: False si 10 no esta en la lista
indice = mi_lista.index("platano") # devuelve posicion de la primera aparicion
mi_lista.sort()                     # ordena la lista
nueva_lista = sorted(mi_lista)      # Devuelve una nueva lista ordenada, sin
modificar la original
mi_lista.reverse()                  # invertir la lista
print(mi_lista.count("pera"))       # devuelve las apariciones de un elementos
mi_lista_copy = mi_lista.copy()     # copiar una lista
mi_lista_copy = mi_lista[:]         # igual que la anterior
mi_lista_anidada = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]  # listas de listas :-)
lista3 = lista1 + lista2            # concatena dos listas en una
lista_repetida = lista1 * 3         # Repetición de la lista1 tres veces
x,y,z = lista                       # saca elementos de la lista a variables
mi_lista_sin_duplicados = list(set(mi_lista)) # elimina duplicados
resultado = all(x % 2 == 0 for x in mi_lista) # devuelve true si todos son pares
resultado = any(x % 2 == 0 for x in mi_lista) # devuelve true si alguno es par
(otros: zip, map, filter, …)
for e in lista: print(e)            # recorrer una lista
for index, l in enumerate(lista): print(index, l) #acompañar indice en recorrido
type(lista1) is list                # comprueba si una variables es una lista
isinstance(lista1, list)            # comprueba si una variables es una lista
```

Python List comprehension o Listas por comprensión

La **list comprehension** en Python es una forma **concisa** y **eficiente** de crear listas a partir de iterables, como listas, tuplas, rangos, etc. Su sintaxis es más compacta que usar un bucle for tradicional y, en muchos casos, más rápida.

Sintaxis básica

```
nueva_lista = [expresion for elemento in iterable if condicion]
```

- `expresion`: la operación o transformación que se aplica a cada elemento.
- `elemento`: la variable que toma los valores del iterable.
- `iterable`: cualquier objeto iterable (lista, tupla, rango, etc.).
- `if condicion` *(opcional)*: filtra los elementos que se incluirán en la lista.

Ejemplos de `List Comprehension`

1. Crear una lista de cuadrados

```
cuadrados = [x**2 for x in range(10)]
print(cuadrados)
# Salida: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Usando un bucle for:

```
cuadrados = []
for x in range(10):
    cuadrados.append(x**2)
```

2. Filtrar números pares

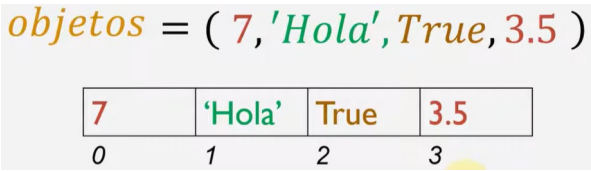
```
pares = [x for x in range(10) if x % 2 == 0]
print(pares)
# Salida: [0, 2, 4, 6, 8]
```

3. Convertir palabras a mayúsculas

```
palabras = ["hola", "mundo", "python"]
mayusculas = [palabra.upper() for palabra in palabras]
print(mayusculas)
# Salida: ['HOLA', 'MUNDO', 'PYTHON']
```

1.2. Tuplas

Las **tuplas** son como las listas pero una vez declaradas, no se pueden modificar (inmutable)




```
monthsOfYear = (“Jan”, “Feb”, “Mar”, “Apr”, “May”, “Jun”, “Jul”, “Aug”, “Sep”,
“Oct”, “Nov”, “Dec”)
monthsOfYear[1]        # acceso al segundo elemento  ‘Hola’
monthsOfYear[-1]       # acceso al ultimo elemento  3.5
monthsOfYear[:2]       # slicing  (7,’Hola’)
monthsOfYear[0] = 8    # ERROR, no podemos modificar
tupla1 = ()            # tupla vacia
```

```
tupla2 = (6,)          # tupla de un solo elemento, hay que poner la coma !!
tupla1 + tupla2        # concatenar tuplas
tupla2 * 3             # concantena una tupla 3 veces
“May” in monthsOfYear  # devuelve True
len(tupla2)            # devuelve el num de elementos de una tupla
max,min,sum            # si los datos son compatibles devuelve el max, min o sum
tupla1.count(23)       #contar ocurrencias de un valor
tupla1.index(23)       # posicion de un valor
tupla1.index(23,3)     # posicion de un valor, empezando a contar desde la pos 3
tupla = tuple(lista)   # Convierte una lista en tupla
sorted(tupla)          # devuelve una lista ordenada
nueva_lista = list(tupla)  # Convierte una tupla en lista
tupla = (1, 2, 3)      # desempaquetado de una tupla
a, b, c = tupla        # a sera 1, b sera 2, c sera 3
tupla = (1, 2, 3, 4, 5)      TUPLA2 = 1,2,3,4    ##  otra forma de definir
a, *b, c = tupla       # a sera 1, c sera 5, b sera [2,3,4] <-lista
v = 1   # NO ES TUPLA    X=(2)   # NO ES TUPLA
Y = (1, )              #  SI ES TUPLA
tupla = 1, 2, ('a', 'b'), 3    # las tuplas también pueden ser anidadas
for t in tupla: print(t) #1, 2, 3    # recorrer una tupla
type(tupla2) is tuple    # comprueba si una variable es una tupla
```

¿Cuándo usar tuplas en lugar de listas?

- Cuando los datos no deben cambiar (ejemplo: coordenadas, días de la semana).
- Cuando se busca mayor eficiencia (las tuplas consumen menos memoria y son más rápidas que las listas).
- Cuando se necesita que los datos sean hashables (las tuplas pueden usarse como claves en diccionarios, mientras que las listas no).

`Tuple comprehension ??`

En python **no** hay tuple comprehension. (pero se puede simular). Si utilizamos () para intentar generar una tuple comprehensión, Python interpretará () **como un `generator expression`**.

```
tupla = (x**2 for x in range(5))
print(tupla)
# Salida: <generator object <genexpr> at 0x...>  (No es una tupla)
```

Esto no crea una tupla, sino un generador (`generator object`).

Si realmente queremos una **tupla**, podemos convertir el generador con **tuple()**

```
tupla = tuple(x**2 for x in range(5))
print(tupla)
```

Ventajas del Generador

1. **Eficiencia de memoria**: No almacena todos los elementos en la memoria, los genera sobre la marcha.
2. **Rendimiento**: Útil para manejar grandes volúmenes de datos sin sobrecargar la memoria.
3. **Simplicidad**: Código más limpio en comparación con definir una función generadora.

1.3. Conjuntos

```
s = {5, 4, 6, 8, 8, 1}
print(s)       #{1, 4, 5, 6, 8}
print(type(s)) #<class 'set'>
```

Características principales de los conjuntos

1. **Elementos únicos**: Un conjunto no puede contener duplicados.
2. **No ordenados**: Los elementos no tienen un orden específico.
3. **Mutables**: Puedes agregar o eliminar elementos de un conjunto.
4. **Elementos inmutables** (podemos añadir y quitar , pero no modificar un elemento)

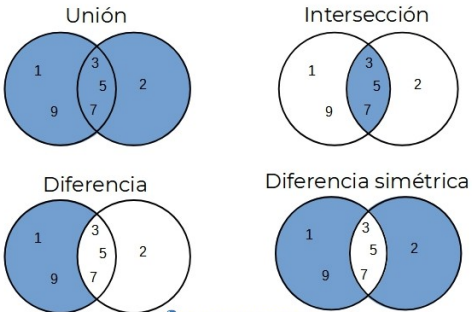
```
conjunto_vacio = set()
conjunto = {1, 2, 2, 3} print(conjunto)   # Salida: {1, 2, 3}
conjunto = set([1, 2, 3, 3])   # convertir lista en conjunto
conjunto.add(3)   remove()    discard()    pop()->elemento aleatorio    clear()
                  #remove da error si no existe el elemento, discard no da error
len(conjunto)    # cantidad de elementos
print ( 5 in conjunto )    # pertenece a , si o no
type(conjunto) is set     # comprueba si es conjunto

a = frozenset([1, 2, 3])   # deja un conjunto inmutable
conjunto = {1, 2.5, "hola", (1, 2), frozenset([3, 4])}

for ss in s: print(ss) #8, 5, 6, 7

print(a.issubset(b))    print(a.issuperset(b))   print(a.isdisjoint(b))

Unión (| o .union())
Intersección (& o .intersection())
Diferencia (- o .difference())
Diferencia simétrica (^ o .symmetric_difference())
```



`El resultado de esta operaciones devuelve un nuevo conjunto.`

```
print ( s1 | s2)
print ( s1.union(s) )
```

El método `update()` en conjuntos (`set`) de Python se usa para agregar elementos de otro iterable (como otro conjunto, lista o tupla) al conjunto original. (no devuelve un nuevo conjunto, sinó que modifica el original)

```
set1.update(iterable)
```

También estan las funciones `set1.intersection_update(iterable)` `.difference_update()` `.symmetric_difference_update()`

Set Comprehension

```
nuevo_set = {expresion for elemento in iterable if condicion}
```

- `expresion` → Lo que se agregará al conjunto.
- `elemento` → Cada ítem tomado del `iterable`.
- `iterable` → Puede ser una lista, tupla, diccionario, conjunto, rango, cadena de texto, etc.
- `if condicion` *(opcional)* → Filtra los elementos antes de agregarlos.

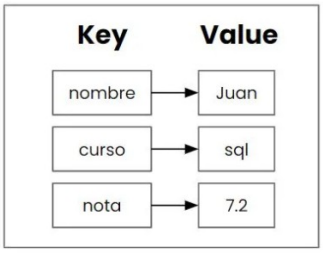
Ejemplo: `pares = {x for x in range(10) if x % 2 == 0}`

1.4. Diccionarios

Los diccionarios almacenan pares de clave-valor.

```
miDiccionario =

{ "nombre": "Juan",
 "curso": "sql",
 "nota": 7.2 }
```



Características principales de los diccionarios

1. **Mutables**
2. **No permiten claves duplicadas.** Si se agrega una clave repetida, su valor será sobrescrito.
3. **Las claves deben ser inmutables.** Pueden ser str, int, tuple, pero no list o dict.
4. **Mantienen el orden de inserción**

```
dic={}  #  Diccionario vacio
persona = {"nombre": "Carlos", "edad": 30, "ciudad": "Madrid"}
print(persona["nombre"])  # Accede a un valor con una clave.  Imprime 'Carlos'
dic["nueva_clave"] = "nuevo_valor"  # Agregar una nueva clave
dic["clave1"] = "nuevo_valor"      # Modificar un valor existente
valor = dic["clave1"]              # Accede al valor de una clave. Si ∄ error
valor = dic.get("clave1", "por defecto")  # Usa get para evitar errores si la
clave no existe
del dic["clave1"]                  # Eliminar una clave específica
valor = dic.pop("clave2", None)    # Elimina una clave y devuelve su valor (o un
valor por defecto)
dic.clear()                        # Vacía el diccionario completamente
if "clave1" in dic:                # Verifica si una clave existe
    print("Clave encontrada")
for clave in dic:                  # Iterar sobre las claves
    print(clave)

for valor in dic.values():         # Iterar sobre los valores
    print(valor)

for clave, valor in dic.items():   # Iterar sobre pares clave-valor
    print(clave, valor)
tamaño = len(dic)                  # Número de elementos en el diccionario
nuevo_dic = dic.copy()             # Crea una copia superficial del diccionario
dic.update({"clave3": "valor3", "clave4": "valor4"})  # Agrega o modifica varias
claves a la vez
claves = dic.keys()      # Obtiene todas las claves como un objeto dict_keys
valores = dic.values()   # Obtiene todos los valores como un objeto dict_values
items = dic.items()      # Obtiene pares clave-valor como un objeto dict_items
type(persona) is dict    # comprueba si es diccionario
dic1 | dic2     #v3.9+ fusiona dos diccionarios, en claves iguales guarda última
dic1 |= dic2   # fusiona y guarda en dic1
```

Orden de los Elementos

Desde Python 3.7, los diccionarios mantienen el orden de inserción de sus elementos. Esto significa que si agregas claves en un orden específico, ese orden se respetará cuando iteres sobre el diccionario:

Valores Mutables

Los valores en un diccionario pueden ser de cualquier tipo, incluidos tipos mutables como listas o incluso otros diccionarios:

```
# Aunque las claves generalmente son cadenas o números, puedes usar cualquier
tipo inmutable como clave, incluyendo tuplas:
dic = {(1, 2): "punto", (3, 4): "otro punto"}
print(dic[(1, 2)])  # "punto"
```

Dictionary Comprehension

Python permite crear listas, conjuntos y diccionarios de forma compacta con **comprensiones**.

```
# Crear un diccionario a partir de una lista (iterable)
numeros = [1, 2, 3, 4]
cuadrados = {n: n**2 for n in numeros}
print(cuadrados) # {1: 1, 2: 4, 3: 9, 4: 16}
```

Resumen de comprensiones (Comprehensions)

```
nueva_lista = [expresion for elemento in iterable if condicion]
gen_expre = (expresion for elemento in iterable if condicion)
nueva_tupla = tuple(expresion for elemento in iterable if condicion)
nuevo_set = {expresion for elemento in iterable if condicion}
nuevo_diccionario = {clave: valor for variable in iterable if condicion}
```

lambda

Variables con expresiones **lambda** (funciones anónimas)

```
operaciones = { "suma": lambda x, y: x + y, "resta": lambda x, y: x – y }
print(operaciones["suma"](10, 5))    # imprime  15
```

(operaciones es una varible diccionario, se verá mas tarde)

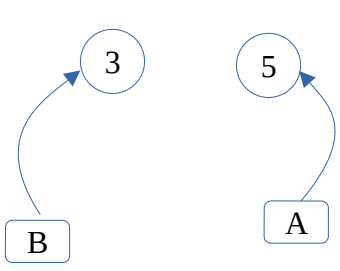
El Garbage Collector (GC) de Python

Es un mecanismo que se encarga de administrar la memoria automáticamente, eliminando objetos que ya no son utilizados para liberar espacio y evitar fugas de memoria.

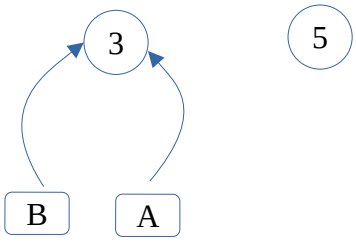
Que ocurre si se asigna un valor a las variables a y b

>>> b=3

>>> a=5



>>> a=b



El objeto "5" se queda sin ser utilizado

Python usa un sistema de conteo de referencias y un recolector de basura basado en generación:

Conteo de referencias: Cada objeto en Python tiene un contador que indica cuántas referencias apuntan a él. Cuando este contador llega a cero, el objeto se elimina automáticamente.

Recolección de basura generacional: Python agrupa los objetos en tres generaciones (0, 1 y

2). Los objetos nuevos comienzan en la generación 0 y, si sobreviven varias ejecuciones del GC, se mueven a generaciones superiores.

Python limpia con más frecuencia los objetos de generación 0 que los de generaciones superiores.

Python permite controlar el recolector de basura menualmete con el módulo gc. Ejem:

```
import gc
gc.collect()  # Forzar la recolección de basura manualmente
gc.get_stats()  # Obtener estadísticas sobre el GC
```

1.5. Funciones avanzadas

Uso de *args y de **kwargs en el pase de parámetros

Uso de *args en Python

El parámetro `*args` permite pasar un número variable de argumentos posicionales a una función en forma de tupla.

Ejemplo básico

```
def sumar(*args):
    return sum(args)

print(sumar(1, 2, 3, 4))  # Output: 10
print(sumar(10, 20))      # Output: 30
```

`*args` recibe todos los valores como una tupla

En Python, **kwargs (abreviatura de *keyword arguments*, argumentos con nombre) se usa en funciones para aceptar un número variable de argumentos con clave-valor.

Uso de **kwargs

Se utiliza en la definición de funciones para recibir argumentos con nombre sin necesidad de especificarlos previamente.

```
def mostrar_info(**kwargs):
    for clave, valor in kwargs.items():
        print(f"{clave}: {valor}")

mostrar_info(nombre="Carlos", edad=30, ciudad="Madrid")
```

Salida:

```
nombre: Carlos
edad: 30
ciudad: Madrid
```

Diferencia con `*args`

- `*args`: Recibe un número variable de argumentos posicionales como una tupla.
- `**kwargs`: Recibe un número variable de argumentos con nombre como un diccionario.

Ejemplo combinando ambos:

```
def ejemplo(*args, **kwargs):
    print("Args:", args)
    print("Kwargs:", kwargs)

ejemplo(1, 2, 3, nombre="Ana", edad=25)
```

Salida:

```
Args: (1, 2, 3)
Kwargs: {'nombre': 'Ana', 'edad': 25}
```

Funciones lambda o expresiones lambda

Las **funciones lambda** en Python son funciones anónimas (es decir, funciones sin nombre) que se pueden definir en una sola línea de código. Se utilizan cuando necesitas una función corta y sencilla, generalmente dentro de otras funciones.

Sintaxis de una función lambda

```
lambda argumentos: expresión
```

- Puede tener múltiples argumentos, separados por comas.
- Solo puede contener una **expresión** (el resultado de esa expresión es lo que devuelve la función).
- No necesita la palabra clave `return`, ya que la expresión se evalúa automáticamente.

Ejemplos:

Sumar dos números

```
suma = lambda x, y: x + y
print(suma(3, 5))  # Salida: 8
```

Elevar un número al cuadrado

```
cuadrado = lambda x: x ** 2
print(cuadrado(4))  # Salida: 16
```

Verificar si un número es par

```
es_par = lambda x: x % 2 == 0
print(es_par(10))  # Salida: True
print(es_par(7))   # Salida: False
```

Usando `map()` para aplicar una función a cada elemento de una lista

```
numeros = [1, 2, 3, 4, 5]
dobles = list(map(lambda x: x * 2, numeros))
print(dobles)  # Salida: [2, 4, 6, 8, 10]
```

Usando `filter()` para filtrar elementos de una lista

```
numeros = [1, 2, 3, 4, 5, 6]
pares = list(filter(lambda x: x % 2 == 0, numeros))
print(pares)  # Salida: [2, 4, 6]
```

Usando `sorted()` para ordenar una lista con una clave personalizada

```
palabras = ["manzana", "banana", "cereza", "kiwi"]
ordenadas = sorted(palabras, key=lambda x: len(x))
print(ordenadas)  # Salida: ['kiwi', 'banana', 'cereza', 'manzana']
```

Decoradores o decorators

Los **decoradores** en Python son una manera elegante y poderosa de modificar el comportamiento de funciones o métodos sin cambiar su código fuente. Son una aplicación del patrón de diseño *decorador*, que permite extender funcionalidades de manera limpia y reutilizable.

¿Qué es un decorador?

Un decorador es simplemente una función que recibe otra función como argumento, la modifica o envuelve con código adicional, y devuelve una nueva función con el comportamiento extendido.

Se utilizan mayormente para:

- **Añadir funcionalidades** sin modificar la función original.
- **Aplicar lógica antes o después de la ejecución** de una función.
- **Validaciones y autenticaciones**, como en frameworks web.
- **Registro de logs** y monitoreo de ejecución.
- **Manejo de caché o repetición de intentos en caso de error**.

Ejemplo básico de un decorador

```
def mi_decorador(func):
    def nueva_funcion():
        print("Antes de llamar la función")
        func()
        print("Después de llamar la función")
    return nueva_funcion

@mi_decorador
def saludo():
    print("¡Hola, mundo!")

saludo()
```

Decoradores con argumentos en la función original

Si la función decorada tiene parámetros, el decorador debe aceptar `*args` y `**kwargs` para permitir cualquier número de argumentos.

```
def decorador_con_args(func):
    def envoltura(*args, **kwargs):
        print(f"Llamando a {func.__name__} con argumentos {args} {kwargs}")
        resultado = func(*args, **kwargs)
        print(f"Resultado: {resultado}")
        return resultado
    return envoltura

@decorador_con_args
def suma(a, b):
    return a + b

suma(3, 5)
```

Salida:

```
Llamando a suma con argumentos (3, 5) {}
Resultado: 8
```

Decoradores con parámetros

Si queremos que el **decorador en sí mismo acepte argumentos**, usamos una función extra.

```
def repetidor(n):
    def decorador(func):
        def envoltura(*args, **kwargs):
            for _ in range(n):
                func(*args, **kwargs)
        return envoltura
    return decorador

@repetidor(3)  # Llamará 3 veces a la función
def hola():
    print("Hola!")

hola()
```

Salida:

```
Hola!
Hola!
Hola!
```

Decoradores predefinidos en Python

Python trae varios decoradores útiles, como:

1. `@staticmethod`: Define un método estático en una clase.
2. `@classmethod`: Permite acceder a la clase dentro del método.
3. `@property`: Convierte un método en una propiedad accesible sin paréntesis.

Ejemplo:

```
class Persona:
    def __init__(self, nombre):
        self._nombre = nombre

    @property
```

```
    def nombre(self):
        return self._nombre

p = Persona("Carlos")
print(p.nombre)  # Se accede como propiedad, sin ()
```

1.6. Ficheros en Python

La función `open()` es parte de la biblioteca estándar de Python, lo que significa que está disponible de forma nativa sin necesidad de importar nada

Abrir un archivo

El método `open()` se utiliza para abrir un archivo. Este método tiene dos argumentos principales:

1. **Nombre del archivo**: El archivo que deseas abrir o crear.
2. **Modo de apertura**: Define cómo deseas interactuar con el archivo (lectura, escritura, etc.).

Modos comunes:

- `"r"`: Leer (por defecto). Lanza un error si el archivo no existe.
- `"w"`: Escribir. Si el archivo existe, lo sobreescribe; si no, lo crea.
- `"a"`: Añadir. Escribe al final del archivo si ya existe; si no, lo crea.
- `"x"`: Crear un archivo nuevo. Lanza un error si el archivo ya existe.
- `"b"`: Modo binario (se combina con los modos anteriores, por ejemplo, `"rb"`).
- `"t"`: Modo texto (por defecto, se combina con `"r"`, `"w"`, etc.).

Ejemplo:

Leer todo el contenido:

```
with open("archivo.txt", "r") as archivo:
    contenido = archivo.read()
    print(contenido)
```

Leer línea por línea:

```
with open("archivo.txt", "r") as archivo:
    for linea in archivo:
        print(linea.strip())  # Elimina saltos de línea al final
```

Leer líneas como una lista:

```
with open("archivo.txt", "r") as archivo:
    lineas = archivo.readlines()
    print(lineas)  # Cada línea será un elemento de la lista
```

Escribir una cadena:

```
with open("archivo.txt", "w") as archivo:
    archivo.write("Hola, mundo\n")
```

Escribir múltiples líneas:

```
lineas = ["Primera línea\n", "Segunda línea\n", "Tercera línea\n"]
with open("archivo.txt", "w") as archivo:
    archivo.writelines(lineas)
```

Añadir contenido a un archivo existente:

```
with open("archivo.txt", "a") as archivo:
    archivo.write("Nueva línea añadida\n")
```

Cosas importantes a tener en cuenta

1. `with` **asegura el cierre del archivo automáticamente.** Es preferible a usar `open()` y `close()`.
2. **Excepciones**: Maneja posibles errores usando bloques `try-except`.

```
try:
    with open("archivo_inexistente.txt", "r") as archivo:
        contenido = archivo.read()
except FileNotFoundError:
    print("El archivo no existe.")
```

3. **Codificación**: Si trabajas con caracteres especiales, especifica la codificación, por ejemplo:

```
with open("archivo.txt", "r", encoding="utf-8") as archivo:
    contenido = archivo.read()
```

Guardar objetos en ficheros

Para guardar y recuperar objetos en un fichero en Python, puedes utilizar el módulo `pickle`, que permite serializar y deserializar objetos. La serialización convierte un objeto en una secuencia de bytes que puede almacenarse en un archivo, y la deserialización lo convierte nuevamente en un objeto usable.

Pasos principales

1. **Serializar (guardar) un objeto**: Usa `pickle.dump()` para guardar el objeto en un archivo.
2. **Deserializar (recuperar) un objeto**: Usa `pickle.load()` para cargar el objeto desde un archivo.

Ejemplo práctico

Guardar un objeto en un fichero

```
import pickle

persona = Persona("Ana", 25)

# Guardar el objeto en un fichero
with open("persona.pkl", "wb") as archivo:  # 'wb' para escritura en modo
binario
    pickle.dump(persona, archivo)

print("Objeto guardado en el archivo 'persona.pkl'.")
```

Recuperar un objeto desde un fichero

```
import pickle

# Leer el objeto desde el fichero
with open("persona.pkl", "rb") as archivo:  # 'rb' para lectura en modo binario
    persona_recuperada = pickle.load(archivo)

print("Objeto recuperado desde el archivo:")
print(persona_recuperada)  # Output: Ana, 25 años
```

Guardar y recuperar múltiples objetos

Puedes guardar múltiples objetos en un archivo serializado, uno tras otro, y luego recuperarlos.

Guardar varios objetos

```
import pickle

# Crear una lista de objetos
personas = [
    Persona("Ana", 25),
    Persona("Luis", 30),
    Persona("María", 28)
]

# Guardar la lista completa en un fichero
with open("personas.pkl", "wb") as archivo:
    pickle.dump(personas, archivo)

print("Lista de objetos guardada en el archivo 'personas.pkl'.")
```

Recuperar varios objetos

```
import pickle

# Leer la lista de objetos desde el fichero
with open("personas.pkl", "rb") as archivo:
    personas_recuperadas = pickle.load(archivo)

print("Objetos recuperados:")
for persona in personas_recuperadas:
    print(persona)
```

Consideraciones importantes

1. **Modo binario**:
   - Usa `"wb"` para escribir en binario y `"rb"` para leer en binario.
2. **Compatibilidad**:
   - Los objetos serializados son específicos de la versión de Python y de la estructura de clases. Cambiar la clase después de guardar los objetos puede causar errores al recuperarlos.
3. **Seguridad**:
   - **No cargues archivos `pickle` de fuentes no confiables**, ya que podrían ejecutar código malicioso.
4. **Serialización en JSON**:

- Si los objetos son simples (como diccionarios o listas), puedes usar `json` en lugar de `pickle`. Sin embargo, JSON no soporta objetos complejos directamente.
    - `import json  json.dump()  ... queda fuera de este curso...`

1.7. Fechas en Python

Python incluye de manera nativa herramientas para el tratamiento de fechas y tiempos mediante el módulo estándar datetime. Este módulo viene integrado en Python y no necesitas instalar ninguna librería adicional para usarlo.

Importar el módulo `datetime`

```
from datetime import datetime, timedelta, date
```

Obtener la fecha y hora actual

```
# Fecha y hora actual
ahora = datetime.now()
print("Fecha y hora actual:", ahora)

# Solo la fecha actual
hoy = date.today()
print("Fecha actual:", hoy)
```

Crear una fecha u hora específica

```
# Crear una fecha específica
mi_fecha = datetime(2025, 1, 28)  # Año, mes, día
print("Fecha específica:", mi_fecha)

# Crear una fecha y hora específica
mi_fecha_hora = datetime(2025, 1, 28, 14, 30, 0)  # Año, mes, día, hora, minuto,
segundo
print("Fecha y hora específica:", mi_fecha_hora)
```

Formato de fechas (`strftime`)

Puedes convertir una fecha u hora en un formato específico como texto:

```
formato = ahora.strftime("%d/%m/%Y %H:%M:%S")
print("Formato personalizado:", formato)

# Ejemplos de códigos de formato:
# %Y - Año completo (2025)
# %m - Mes en número (01)
# %d - Día (28)
# %H - Hora en formato 24 horas
# %I - Hora en formato 12 horas
```

```
# %M - Minuto
# %S - Segundo
# %p - AM o PM
```

Convertir texto en fecha (`strptime`)

Puedes convertir una cadena en un objeto `datetime`:

```
texto = "28/01/2025 14:30:00"
fecha_convertida = datetime.strptime(texto, "%d/%m/%Y %H:%M:%S")
print("Texto convertido a fecha:", fecha_convertida)
```

Operaciones con fechas

1. **Sumar o restar tiempo usando `timedelta`:**

```
# Sumar 10 días a la fecha actual
nueva_fecha = ahora + timedelta(days=10)
print("Fecha con 10 días más:", nueva_fecha)

# Restar 5 horas
nueva_hora = ahora - timedelta(hours=5)
print("Fecha con 5 horas menos:", nueva_hora)
```

2. **Calcular la diferencia entre dos fechas:**

```
fecha1 = datetime(2025, 1, 28)
fecha2 = datetime(2025, 2, 5)
diferencia = fecha2 - fecha1
print("Días de diferencia:", diferencia.days)
```

Comparar fechas

Puedes usar operadores de comparación como <, >, == para comparar objetos de fecha y hora:

```
if fecha1 < fecha2:
    print("Fecha1 es anterior a Fecha2")
else:
    print("Fecha1 es posterior o igual a Fecha2")
```

Otros modulos relacionados con fechas son time y calendar

Módulo `time`

`El módulo time se utiliza para trabajar con el tiempo de sistema y realizar operaciones como medir intervalos o pausas.`

Características principales:

- Manejo de timestamps (segundos desde el **Epoch**, 1 de enero de 1970).
- Realizar pausas en la ejecución del programa.

Ejemplo básico:

```
import time

# Pausar la ejecución por 2 segundos
time.sleep(2)

# Obtener el timestamp actual
timestamp = time.time()
print("Timestamp actual:", timestamp)
```

Módulo `calendar`

Este módulo es útil para trabajar con calendarios y verificar propiedades de fechas (como días de la semana, años bisiestos, etc.).

Características principales:

- Generar calendarios en texto o HTML.
- Verificar si un año es bisiesto.
- Obtener el primer día de la semana y número de días en un mes.

Ejemplo básico:

```
import calendar

# Verificar si un año es bisiesto
print("¿2024 es bisiesto?", calendar.isleap(2024))

# Obtener calendario de enero 2025
print(calendar.month(2025, 1))
```

Módulo `zoneinfo` (Python 3.9+)

Este módulo permite manejar zonas horarias de manera más sencilla y eficiente.

Características principales:

- Trabajar con zonas horarias específicas.
- Convertir tiempos entre zonas horarias.

Ejemplo básico:

```
from datetime import datetime
from zoneinfo import ZoneInfo

# Definir zonas horarias
utc = datetime.now(ZoneInfo("UTC"))
madrid = datetime.now(ZoneInfo("Europe/Madrid"))

print("Hora UTC:", utc)
print("Hora Madrid:", madrid)
```

Librerías externas (opcional)

Aunque Python incluye herramientas robustas para manejar fechas, a veces puedes necesitar librerías externas para casos más complejos. Las más populares son: `dateutil y arrow`

1.8. Trabajar con arrays

En Python, se pueden manejar arrays utilizando listas o el módulo numpy Necesitaremos una biblioteca externa que nos proporcionara funcinalidades específicas **`.NumPy.`**

NumPy `(Numerical Python)` es una de las librerías más fundamentales y populares en Python para trabajar con **computación científica** y **manipulación de datos numéricos.** Es ampliamente utilizada en áreas como análisis de datos, inteligencia artificial, machine learning, procesamiento de imágenes y más.

¿Qué es NumPy?

NumPy es una biblioteca que proporciona:

1. **Arrays multidimensionales** para trabajar con datos de manera más eficiente que las listas de Python.
2. Funciones matemáticas optimizadas para operaciones rápidas y vectorizadas.
3. Herramientas para manipulación de datos, álgebra lineal, transformadas de Fourier, generación de números aleatorios, y más.

Instalación

Si no tienes NumPy instalado, puedes hacerlo con:

```
$ pip3 install numpy
```

Características principales de NumPy

Arrays

El tipo de datos principal de NumPy es el **ndarray** (N-dimensional array). Estos son más rápidos y ocupan menos memoria que las listas de Python.

Ejemplo básico:

```
import numpy as np

# Crear un arreglo unidimensional
arreglo = np.array([1, 2, 3, 4])
print("Arreglo:", arreglo)

# Crear un arreglo bidimensional (matriz)
matriz = np.array([[1, 2, 3], [4, 5, 6]])
print("Matriz:\n", matriz)
```

Vectorización

NumPy permite realizar operaciones matemáticas en todos los elementos de un arreglo de una sola vez (sin bucles).

Ejemplo:

```
# Operaciones matemáticas en arrays
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
print("Suma:", a + b)  # [5 7 9]
print("Producto escalar:", a * 2)  # [2 4 6]
```

Ahorro de memoria

NumPy almacena los datos en un tipo homogéneo, lo que lo hace más eficiente que las listas:

```
import sys

lista = range(1000)
print("Tamaño de lista:", sys.getsizeof(1) * len(lista))

arreglo = np.arange(1000)
print("Tamaño de arreglo NumPy:", arreglo.nbytes)
```

Funciones útiles

NumPy incluye muchas funciones predefinidas para cálculos matemáticos:

- **Aritméticas:** `np.sum`, `np.mean`, `np.median`, `np.std`, `np.var`.
- **Operaciones lógicas:** `np.all`, `np.any`.
- **Generación de números aleatorios:** `np.random.rand`, `np.random.randint`.
- **Álgebra lineal:** `np.dot`, `np.linalg.inv`, `np.linalg.eig`.

Ejemplo:

```
# Estadísticas básicas
datos = np.array([10, 20, 30, 40])
print("Media:", np.mean(datos))
print("Desviación estándar:", np.std(datos))
```

Creación de arrays especializados

NumPy facilita la creación de arrays con valores específicos:

```
# Arreglo de ceros
ceros = np.zeros((2, 3))  # 2 filas, 3 columnas
print("Ceros:\n", ceros)

# Arreglo de unos
unos = np.ones((3, 2))
print("Unos:\n", unos)

# Arreglo con valores en un rango
rango = np.arange(0, 10, 2)  # Inicio, fin, paso
print("Rango:", rango)

# Arreglo de números espaciados uniformemente
espaciados = np.linspace(0, 1, 5)  # Inicio, fin, cantidad
print("Espaciados uniformemente:", espaciados)
```

Indexación y slicing avanzados

Puedes acceder y modificar elementos, incluso usando condiciones:

```
arreglo = np.array([10, 20, 30, 40, 50])

# Acceso por índices
print("Elemento en índice 2:", arreglo[2])  # 30

# Slicing
print("Subarreglo:", arreglo[1:4])  # [20 30 40]

# Condicionales
print("Elementos mayores a 25:", arreglo[arreglo > 25])  # [30 40 50]
```

Operaciones con matrices

NumPy es muy eficiente para realizar cálculos con matrices:

```
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])

# Producto punto
print("Producto punto:\n", np.dot(A, B))

# Transposición
print("Transpuesta de A:\n", A.T)
```

Números aleatorios

Puedes generar valores aleatorios con facilidad:

```
# Número aleatorio entre 0 y 1
aleatorio = np.random.rand()
print("Número aleatorio:", aleatorio)

# Arreglo aleatorio entero
aleatorios = np.random.randint(1, 10, size=(3, 3))
print("Matriz aleatoria:\n", aleatorios)
```

¿Por qué usar NumPy?

1. **Velocidad:** NumPy es hasta 50 veces más rápido que las listas normales de Python para operaciones numéricas.
2. **Optimización de memoria:** Usa menos memoria que las estructuras estándar de Python.
3. **Facilidad de uso:** Ofrece muchas funciones matemáticas, estadísticas y de álgebra lineal listas para usar.
4. **Interoperabilidad:** Es compatible con otras librerías populares como **Pandas**, **Matplotlib**, **Scikit-learn**, y más.

Otras bibliotecas en Python

Incluidas

`os` Para interactuar con el sistema operativo.

`sys` Permite interactuar con el intérprete de Python y sus argumentos.

`json` Para trabajar con datos en formato JSON.

`datetime` Para trabajar con fechas y tiempos.

`math` Incluye funciones matemáticas avanzadas

`decimal` Si necesitas realizar cálculos con mayor precisión que la que ofrecen los floats

`random` Generación de números aleatorios

`re` Para trabajar con expresiones regulares.

`collections` Incluye tipos de datos avanzado

`itertools` Herramientas para trabajar con iteradores

`sqlite3` Librería incorporada para manejar bases de datos SQLite directamente desde Python

Podemos ver los módulos incluidos desde python con `help()` y luego `help> modules`

Externas

`NumPy` Para cálculos numéricos rápidos y eficientes

`Pandas` Manejo de datos tabulares (filas y columnas

`Matplotlib` Creación de gráficos y visualización básica.

`Seaborn` Extensión para gráficos estadísticos más atractivos.

`Scikit-learn` Librería para machine learning.

`Flask y Django` Frameworks para desarrollo web

`Requests` Para realizar peticiones HTTP (GET, POST, etc.)

`BeautifulSoup` Librería para web scraping

`TensorFlow y PyTorch` Librerías avanzadas para inteligencia artificial y deep learning

`typer` Creación rápida de CLIs en Python.

`OpenCV` Procesamiento de imágenes y visión por ordenador

`Pillow` Librería para manipulación de imágenes

y mas.....

2. Programación OO en Python

Un **objeto** en Python es una instancia de una **clase** y tiene:

- **Atributos**: Datos que describen las propiedades del objeto.
- **Métodos**: Funciones que definen el comportamiento del objeto.

Por ejemplo:

```
class Perro:
    def __init__(self, nombre, raza):
        self.nombre = nombre
        self.raza = raza

    def ladrar(self):
        print(f"{self.nombre} dice: ¡Guau!")

# Crear un objeto (instancia) de la clase
mi_perro = Perro("Fido", "Labrador")

# Acceder a los atributos y métodos
print(mi_perro.nombre)  # Output: Fido
mi_perro.ladrar()       # Output: Fido dice: ¡Guau!
```

Una **clase** es la plantilla o el modelo que define cómo se crean los objetos.

Crear una clase

```
class Coche:
    def __init__(self, marca, modelo, color):
        self.marca = marca
        self.modelo = modelo
        self.color = color

    def describir(self):
        print(f"Este coche es un {self.color} {self.marca} {self.modelo}.")
```

Crear objetos (de una clase)

```
mi_coche = Coche("Toyota", "Corolla", "rojo")
otro_coche = Coche("Honda", "Civic", "azul")

# Acceder a atributos y métodos
print(mi_coche.marca)   # Output: Toyota
mi_coche.describir()    # Output: Este coche es un rojo Toyota Corolla.
```

Atributos de instancia vs. clase

- **Atributos de instancia**: Propiedades específicas de un objeto (definidas en `__init__`).
- **Atributos de clase**: Compartidos entre todos los objetos de la clase.

```
class Animal:
    especie = "Mamífero"  # Atributo de clase

    def __init__(self, nombre):
        self.nombre = nombre  # Atributo de instancia
```

```
gato = Animal("Gato")
perro = Animal("Perro")

print(gato.especie)  # Output: Mamífero (compartido)
print(gato.nombre)   # Output: Gato (específico del objeto)
```

Métodos especiales

Los métodos especiales (también llamados "métodos mágicos") comienzan y terminan con __. También se les conoce como “**dunder**”. El término "dunder" viene de **"Double UNDERscore"**

- `__init__`: Inicializa un objeto. Es llamado siempre que se crea un objeto.
- `__str__`: Devuelve una representación en cadena del objeto.
- `__len__`: Devuelve la longitud (útil para implementar en clases personalizadas).

Ejemplo:

```
class Persona:
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad

    def __str__(self):
        return f"{self.nombre}, {self.edad} años"

persona = Persona("Ana", 30)
print(persona)  # Output: Ana, 30 años
```

Otros mètodos dunder son:

- `__add__` Sumar objetos `__sub__` restar objetos
- `__eq__ __lt__ __gt_` Comparar objetos
- `__iter__ __next__` Iterar objeto
- `__getattr__ __setattr__ __delattr_` Acceso a atributos de objetos

Para ver los métodos de un objeto : `print(dir(obj))`

Herencia

Las clases pueden heredar de otras para reutilizar código.

```
class Vehiculo:
    def __init__(self, marca, modelo):
        self.marca = marca
        self.modelo = modelo

    def describir(self):
        print(f"Vehículo: {self.marca} {self.modelo}")

# Subclase que hereda de Vehiculo
class Coche(Vehiculo):
    def __init__(self, marca, modelo, puertas):
        super().__init__(marca, modelo)  # Llama al constructor de la clase base
        self.puertas = puertas

    def describir(self):
        print(f"Coche: {self.marca} {self.modelo} con {self.puertas} puertas.")
```

```
coche = Coche("Ford", "Fiesta", 4)
coche.describir()  # Output: Coche: Ford Fiesta con 4 puertas.
```

Encapsulación

Puedes controlar el acceso a los atributos con modificadores de acceso:

- **Público** (`nombre`): Accesible desde cualquier lugar.
- **Protegido** (`_nombre`): Indica que es interno (por convención).
- **Privado** (`__nombre`): Sólo accesible dentro de la clase.

```
class CuentaBancaria:
    def __init__(self, titular, saldo):
        self.titular = titular
        self.__saldo = saldo  # Atributo privado

    def depositar(self, monto):
        self.__saldo += monto

    def mostrar_saldo(self):
        print(f"Saldo: {self.__saldo}")

cuenta = CuentaBancaria("Carlos", 1000)
cuenta.depositar(500)
cuenta.mostrar_saldo()  # Output: Saldo: 1500
```

Polimorfismo

Permite usar el mismo método en diferentes clases con comportamientos distintos.

```
class Ave:
    def sonido(self):
        print("Canto genérico")

class Canario(Ave):
    def sonido(self):
        print("Trinar")

class Pato(Ave):
    def sonido(self):
        print("Cuac cuac")

animales = [Canario(), Pato(), Ave()]
for animal in animales:
    animal.sonido()
```

Clases iterables

Puedes crear tus propios iterables definiendo una clase que implemente los métodos `__iter__() y __next__()`. Esto te permite crear secuencias personalizadas que se pueden recorrer en bucles for.

Ejemplo de un iterable personalizado:

```
class Contador:
  def __init__(self, inicio, fin):
    self.inicio = inicio
```

```
    self.fin = fin

  def __iter__(self):
    self.numero = self.inicio
    return self

  def __next__(self):
    if self.numero <= self.fin:
      resultado = self.numero
      self.numero += 1
      return resultado
    else:
      raise StopIteration

contador = Contador(1, 5)

for numero in contador:
  print(numero)
```

En este ejemplo, la clase `Contador` es un iterable que genera números del 1 al 5. El bucle `for` itera sobre cada número generado por el iterable y lo imprime en la consola.

3. Ejercicios

Ejercicios para practicar:

1. **Tabla de multiplicar:** Crea una lista con las multiplicaciones del número 5 (del 1 al 10).
2. **Tabla de multiplicar genérica:** Crea una lista con las multiplicaciones de un numero dado (del 1 al 10).
3. **Diccionario de personas:** Crea un diccionario con el nombre y la edad de varias personas y luego imprímelo.

Ejercicios Básicos

1. **Suma de números en una lista:** Crea una función que reciba una lista de números y devuelva la suma de todos ellos.
2. **Número mayor en una lista:** Crea un programa que reciba una lista de números y devuelva el mayor número de la lista.
3. **Fibonacci:** Crea una función que reciba un número `n` y devuelva los primeros `n` números de la secuencia de Fibonacci.

Ejercicios Intermedios

6. **Ordenar una lista de tuplas:** Dada una lista de tuplas con dos elementos (nombre, edad), ordena la lista según la edad de las personas.

```
personas = [("Carlos", 30), ("Ana", 25), ("Luis", 35)]
```

El resultado debe ser la lista ordenada por edad de menor a mayor.

Ejercicios Avanzados

11. **Buscar un elemento en una lista:** Crea una función que busque un valor en una lista y devuelva su índice si lo encuentra. Si no lo encuentra, debe devolver "No encontrado".
12. **Matriz transpuesta:** Crea una función que reciba una matriz (lista de listas) y devuelva su transpuesta. La transpuesta de una matriz es una nueva matriz donde las filas y columnas se intercambian.

```
matriz = [[1, 2], [3, 4]]
# La transpuesta debería ser:
# [[1, 3], [2, 4]]
```

13. **Generador de contraseñas aleatorias:** Crea una función que genere una contraseña aleatoria de una longitud especificada. La contraseña debe contener al menos una letra mayúscula, una letra minúscula, un número y un carácter especial.

14. **Eliminar duplicados de una lista:** Crea una función que reciba una lista con elementos duplicados y devuelva una nueva lista con los elementos únicos (sin duplicados).

15. **Encuentra los números primos:** Crea una función que reciba un número `n` y devuelva una lista con todos los números primos menores o iguales a `n`.

16. **Simulación de un sistema de calificación:** Crea un programa que permita ingresar las calificaciones de varios estudiantes y luego calcule el promedio de todas las calificaciones, además de mostrar las calificaciones por encima o debajo del promedio.

Ejercicios con Bibliotecas Externas

21. **Uso de `random`:** Usa la biblioteca `random` para generar una lista de 10 números aleatorios entre 1 y 100, y luego encuentra el número más grande y el más pequeño.

22. **Trabajo con fechas (`datetime`):** Crea un programa que reciba una fecha (en formato `DD/MM/YYYY`) y calcule cuántos días han pasado desde esa fecha hasta hoy.

23. **Leer un archivo:** Crea un programa que lea un archivo de texto línea por línea e imprima cada línea en la consola.
